UNIVERSIDAD NACIONAL AUTONOMA
DE MEXICO

FACULTAD DE INGENIERIA

IMPLEMENTACION DE UN SISTEMA MULTIAGENTES PARA INTERNET MEDIANTE EL USO DE TECNICAS DE INTELIGENCIA ARTIFICIAL DISTRIBUIDA

T E S I S

QUE PARA OBTENER EL TITULO DE:

INGENIERIA EN COMPUTACION

P R E S E N T A :

ELOHIM PUON SANCHEZ

DIRECTOR: ING. NICOLAS KEMPER VALVERDE

MEXICO, D. F. 1997

Tabla de Contenido

Parte I, Planteamiento del Problema

Vivimos un momento en el que pocas cosas están a la medida para ser controladas por el hombre de manera individual, motivo por el cual actualmente las personas tienden a atacar de forma colectiva los problemas que una sola persona sería incapaz de dominar. Es ése el ambiente en el cual ha surgido Internet, como una herramienta que facilita el intercambio de ideas y conocimientos entre personas cualesquiera sin importar su localización geográfica, posición social, edad o sexo.

Sin embargo, el número de usuarios conectados a la red se ha incrementado de forma acelerada gracias a la cada vez mayor accesibilidad a computadoras y a las redes. De igual manera, el número de host conectados a Internet de forma directa, crece exponencialmente, ocasionando el crecimiento del volumen de información disponible a través de una computadora conectada, y la rapidez con que la información disponible se actualiza o desaparece. Esto explica la desorientación que las personas sienten mientras navegan por la red, y lo cierto es que sólo una persona por lo menos medianamente iniciada y capacitada puede utilizar de una manera eficaz éste recurso, de ahí que una gran parte de los usuarios de Internet sean personas de una u otra manera relacionadas con la ciencia de la computación.

Para hacer de Internet un servicio realmente accesible a cualquier persona, debemos asegurar su facilidad de uso y garantizar la obtención de resultados satisfactorios incluso para usuarios nuevos o inexpertos. Una persona no debe perderse entre montañas de información de poco interés antes de encontrar la pequeña porción que le interesa.

La solución propuesta en éste trabajo consiste en un sistema altamente personalizado que trabaja exclusivamente para satisfacer los requerimientos de información del usuario, y dado que la red en sí no provee ninguna herramienta que trate a cada usuario atendiendo sus intereses individuales, se hace necesario contar con aplicaciones propias y personales que además de poder descubrir e interpretar información en la red, puedan aprender las necesidades cambiantes del dueño y adaptarse continuamente a ellas. Éstas aplicaciones son Agentes.

La implementación de las redes de computadoras fue una gran paso en la construcción de sociedades en donde la colaboración entre los individuos requiere que las vías de comunicación sean establecidas y utilizadas de forma efectiva. La Inteligencia Artificial Distribuida (IAD) es una rama de la inteligencia artificial que investiga modelos de conocimiento y técnicas de comunicación y razonamiento que los

agentes computacionales pueden necesitar para participar en una sociedad compuesta de computadoras, personas y otros agentes, es decir, investiga situaciones en donde varios sistemas interactúan para resolver un problema común.

Éste trabajo propone la utilización de un sistema multiagentes como una solución para realizar una tarea que un solo agente podría encontrar muy difícil, tanto por la diversidad de las tareas necesarias como por la naturaleza distribuida de la red. Un sistema multiagentes es una colección de agentes autónomos que trabajan para resolver un problema común que queda mas allá de sus capacidades individuales. Es por ese motivo que nuestro sistema multiagentes constituye un campo de aplicación de la IAD.

Parte II, Fundamentos

2.1. Fundamentos de la Inteligencia Artificial Distribuida

2.1.1. ¿Qué es un Agente ?

Un agente es un sistema situado dentro de, y que es parte de, un ambiente, y que es capaz de percibirlo mediante sensores y actuar sobre él mediante efectores (figura 2.1).

Se dice que un agente es racional si se espera que las acciones que emprenda maximicen su éxito, basándose en la evidencia provelda por su secuencia de percepciones y el conocimiento que el agente posea [21].

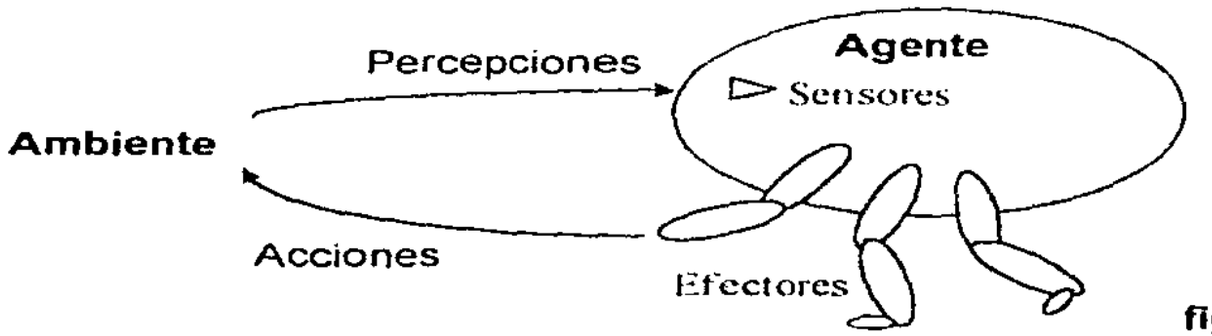


figura 2.1

La complejidad de un agente es determinada por la naturaleza del ambiente en que éste habitará, por lo tanto podemos distinguir agentes racionales con diversos grados de complejidad. La manera mas sencilla de construir un agente sería representándolo mediante una tabla, en la que para cada posible secuencia de percepción se tiene la acción correcta a realizar. Por el tamaño de la tabla y por que es necesario que el constructor del agente deposite en ella todo el conocimiento, éste no es un tipo de agente práctico.

En una solución basada en reglas se restringe el contenido de la tabla a aquellas asociaciones entrada/salida que ocurren mas frecuentemente. Un agente que utiliza reglas para decidir cómo actuar, es un agente que actúa por reflejos y en respuesta únicamente a su secuencia de percepciones. Si se desea lograr un comportamiento mas flexible es necesario que el agente cuente con una representación interna del estado del mundo externo. Esto es especialmente útil cuando el agente no puede percibir el ambiente en forma total. El conocimiento del agente sobre el mundo externo comprende dos áreas : conocimiento sobre cómo evoluciona el mundo sin la intervención del agente, y conocimiento sobre cómo las acciones del agente afectan el estado del ambiente.

Además, un agente destinado a la realización de una tarea especifica necesita conocer cuáles son sus objetivos, de manera que a partir de

sus percepciones y conocimiento pueda decidir cuál de varias alternativas posibles de acción es mejor tomar. En ésta decisión el agente debe saber cuál será su condición si ejecuta ciertas acciones y decidir cuál de ellas lo conducirá hacia el logro de sus metas. Además, un agente puede evaluar la conveniencia de varios posibles caminos de acción, compararlos y elegir uno de entre ellos.

Este trabajo se ocupa únicamente de agentes de software, es decir, programas. Existen agentes de software de muy diversas características, por lo que podemos, mas que diferenciar unos de otros, establecer ciertas características que poseen algunos de ellos [26].

Un agente tiene autonomía si puede tomar la iniciativa sin necesidad de la intervención externa por parte del usuario, y controlar de manera no trivial sus propias acciones.

Un agente de software es colaborativo si no obedece ciegamente las órdenes del usuario, sino que tiene la habilidad de modificar las peticiones, volver a preguntar o incluso negarse a satisfacer algunas de ellas

Un agente es flexible si sus acciones no dependen totalmente del conocimiento depositado en él por el creador del agente, sino que puede escoger dinámicamente qué acciones emprender y en qué

secuencia, en respuesta a su percepción pasada y actual del ambiente externo.

Algunos agentes, a diferencia de los programas convencionales, pueden sentir cambios en su ambiente y decidir cuándo actuar y cuándo no.

Los agentes *proactivos* son aquellos que no actúan simplemente en respuesta a estímulos provenientes del ambiente, sino que pueden exhibir comportamiento orientado a metas al tomar ellos mismos la iniciativa.

Es posible que un agente tenga carácter, es decir, una personalidad y un estado emocional bien definidos.

Un agente es comunicativo, si puede entablar una comunicación compleja con otros agentes, incluyendo personas, para obtener información o ayuda para conseguir sus objetivos.

La benevolencia es una característica que tienen los agentes cuando éstos tendrán siempre la voluntad de hacer lo que otros agentes le pidan, debido a que sus metas no entran en conflicto con las metas de ningún otro agente.

La capacidad de adaptación permite a un agente configurarse a sí mismo de acuerdo a los cambios en el ambiente, por ejemplo las preferencias cambiantes de su dueño, basándose en su experiencia previa.

Un agente de software es móvil si puede transportarse de una máquina a otra sin importar si las arquitecturas o plataformas son diferentes

Debido a la diversidad de los problemas en los que se puede aplicar un sistema basado en agentes, aún no se logra un consenso sobre la importancia relativa de cada una de las características mencionadas ; sin embargo, se ha convenido que las características de movilidad, benevolencia, racionalidad, adaptabilidad y colaboratividad, son las características que distinguen a los agentes de los programas ordinarios

Dado que nuestros agentes serán diseñados para ayudar a usuarios de Internet a encontrar información de su interés en la red, es necesario identificar las características específicas de las tareas que deberán realizar y las del ambiente en que operarán.

Podemos categorizar los diferentes tipos de ambiente en los que un agente puede habitar, de acuerdo a las siguientes características [21]:

Accesible o inaccesible: si el aparato sensor del agente le da acceso al estado completo del ambiente, y por lo tanto detectan cualquier aspecto que sea relevante para escoger una acción, el ambiente es accesible.

Determinístico o no determinístico : si el próximo estado del ambiente está completamente determinado por el estado actual y las acciones escogidas por los agentes, entonces el ambiente es deterministico. Si el ambiente es inaccesible, puede parecer no determinístico, lo cual es particularmente cierto en ambientes complejos, siendo difícil llevar un registro de todos los aspectos inaccesibles.

Episódico o no episódico : en un ambiente episódico, la experiencia del agente está dividida en episodios. Cada episodio consiste en un una secuencia de percepción acción por parte del agente, y un episodio y las acciones tomadas en él, no dependen de episodios anteriores.

Estático o dinámico : si el ambiente puede cambiar mientras el agente se encuentra deliberando, entonces el ambiente es dinámico para ese agente, de otra manera es estático. En un ambiente estático, un agente no necesita seguir sensando el mundo mientras decide su próxima acción, en tanto que en un ambiente estático, el agente no necesita preocuparse por el paso del tiempo.

Discreto o continuo : Si hay un número limitado de percepciones y acciones distintas y claramente definidas, el ambiente es discreto.

2.1.2. Sistemas Multiagentes y el Papel de la Inteligencia Artificial Distribuida

La inteligencia artificial distribuida es una rama de la inteligencia artificial que se ha estado dedicando, por mas de una década, a la investigación de modelos de conocimiento, técnicas de razonamiento y comunicación que los agentes computacionales pudieran necesitar para participar en sociedades compuestas de computadoras y gente. Mas generalmente, la IAD investiga situaciones en las que varios sistemas interactúan para resolver un problema común [22].

La IAD es multidisciplinaria por naturaleza, y se beneficia de avances en varios campos : inteligencia artificial, ciencias sociales y organizacionales, computación distribuida, procesamiento de lenguaje natural, ciencias cognitivas, filosofía, etc. Usualmente se distinguen dos áreas dentro de la inteligencia artificial distribuida, solución distribuida de problemas y sistemas multiagentes.

En la solución distribuida de problemas se considera cómo la tarea de resolver un problema particular puede ser dividida entre un cierto número de módulos, que cooperan compartiendo conocimiento acerca del problema y su solución. En un sistema de solución distribuida de

problemas puro, toda la interacción entre los módulos está incorporada como una parte integral del sistema.

La investigación en sistemas multiagentes se orienta al comportamiento de una colección de agentes autónomos que tratan de resolver un problema. Un sistema multiagentes puede ser definido como una red poco acoplada de agentes que trabajan juntos para resolver un problema que está mas allá de sus capacidades individuales. Estos agentes son autónomos y pueden ser heterogéneos (caracterizados por distintas capacidades de solución de problemas). Los sistemas multiagentes encuentran su aplicación en situaciones que son inherentemente distribuidas, ya sea espacial o funcionalmente.

La inteligencia artificial distribuida abarca una amplia gama de áreas de investigación que pueden ser estudiadas desde diferentes perspectivas. Desde el punto de vista de cada agente, la IAD investiga las categorías de agentes, la estructura y mantenimiento del conocimiento, las habilidades de razonamiento, las habilidades de adaptación y aprendizaje y la arquitectura interna del agente. Desde el punto de vista grupal, la IAD investiga la organización del grupo, la coordinación, la cooperación, la negociación, el comportamiento global coherente, la planeación y la comunicación [22].

Un agente puede caracterizarse por diferentes estructuras de conocimiento : hechos, creencias, metas, intenciones, preferencias, motivaciones, deseos, etc. Usualmente un agente adquiere nuevo conocimiento como el resultado de mensajes enviados por otros agentes o por el ambiente. Por otra parte, los agentes deben razonar acerca de diversos aspectos de la realidad, que pueden incluir el comportamiento de otros agentes y su conocimiento, así como sus propias creencias, deseos e intenciones. Un agente puede incluso explorar diversas hipótesis antes de tomar una decisión. Un agente puede también necesitar adaptarse a su ambiente y al comportamiento de otros agentes.

Un sistema multiagentes puede definirse en términos de las relaciones entre los agentes ; la organización, cooperación, negociación y el rol que desempeña cada agente, son factores que determinan el comportamiento grupal del sistema. Aunque históricamente los sistemas multiagentes tienen muy poca flexibilidad para adaptar su comportamiento global, hoy es crítico desarrollar una manera útil de construir sistemas multiagentes mas adaptables, a partir de algunos conceptos básicos sobre organizaciones y *cambio organizacional*.

Una organización no debe ser concebida como una relación estructural entre un conjunto de agentes, o un conjunto de limitaciones a su conducta establecidas externamente. Debe verse a la organización

como incrustada en las creencias, intenciones y deseos de los agentes mismos. Una organización queda entonces definida como un conjunto de agentes con tareas mutuas, creencias mutuas, y eventualmente intenciones conjuntas cuando estos agentes actúan juntos para alcanzar una meta propuesta.

Cuando las tareas que realiza un agente dependen de los resultados obtenidos por otro agente, es necesario implantar un sistema de coordinación, ya que es necesario sincronizar la comunicación de resultados intermedios

Cuando los intereses de dos o mas agentes entran en conflicto y ponen en peligro el comportamiento cooperativo, se hace evidente la necesidad de un mecanismo de negociación. La negociación es el proceso de mejorar el consenso en puntos de vista o planes comunes, mediante el intercambio estructurado de información relevante.

El trabajo conjunto de los mecanismos mencionados anteriormente permite al sistema multiagentes lograr un comportamiento global coherente. La coherencia global significa que las acciones tomadas por un agente tengan sentido respecto a las metas comunes de todo el grupo. La coherencia global depende ampliamente no solo de la estructura del grupo de agentes, sino también de que cada agente tenga un amplio conocimiento de las metas globales, lo que ya se ha

logrado dentro del grupo, que necesita hacerse a continuación, y las actividades e intenciones de los demás agentes.

Por otra parte, es posible obtener una mejor coordinación si se manipula de cierta manera el comportamiento de los agente mediante la división explícita de labores. Técnicas como la planeación centralizada para múltiples agentes, reconciliación de planes, planeación distribuida y análisis organizacional, facilitan la adecuación de las actividades de los agentes asignándoles tareas después de razonar acerca de las consecuencias de llevar a cabo tales tareas en un orden particular. Un plan multiagentes puede ser generado de manera centralizada, y después descompuesto en submetas o subtareas y distribuido entre el grupo de agentes, o puede ser generado a partir de planes individuales preexistentes sujetos a reconciliación.

Es necesario enfatizar que nada de lo descrito antes sería posible sin la comunicación agente-agente. La comunicación permite a los agentes intercambiar información y coordinar sus actividades. Existen dos estrategias principales por medio de las cuales puede lograrse la comunicación agente-agente. Los agentes pueden intercambiar mensajes directamente o pueden accesar un repositorio compartido de datos en donde la información puede ser pegada y leída por otros agentes. Además, las soluciones posibles al problema de la comunicación varían desde aquellas que no involucran ninguna

comunicación, hasta aquellas basadas en sofisticada comunicación de alto nivel.

2.2. Los Agentes de Software.

2.2.1. ¿Qué es un agente de software ?

Los agentes que nos ocupan son agentes de software, es decir, programas capaces de ser ejecutados en una computadora, y como tales están compuestos de tres partes : código, estado y datos.

En los últimos años se ha realizado una investigación intensiva en lo referente al diseño y construcción de agentes de software, y se han llevado a cabo numerosas y muy variadas implementaciones de agentes destinados a resolver una cantidad igualmente grande y variada de problemas.

La investigación referente a software orientado a agentes se ha conducido por dos caminos diferentes pero convergentes. Por una parte se hace énfasis en la utilización de técnicas de inteligencia artificial para llevar a cabo tareas relacionadas con el manejo de información distribuida y dinámica, y por otro lado, se hace énfasis sobre los agentes como interfaces hombre-máquina que básicamente actúan como interfaces efectivas entre las expectativas del usuario y

las capacidades de la computadora, y cuya función primordial es permitir formas de interacción que serían difíciles de evocar con una interfaz gráfica común [13].

De cualquier forma, un sistema basado en agentes no tiene otro propósito que el de facilitar y hacer mas amigable el uso de servicios de cómputo, sea una sola computadora o una red de computadoras. La utilización de agentes de software es una tecnología orientada a proveer un mayor nivel de abstracción de los procesos que realiza un sistema de cómputo o de información [24].

2.2.2. Algunos tipos de agentes de software.

Podemos clasificar los agentes de software desarrollados a la fecha de acuerdo a las funciones que realizan de la siguiente manera [26]:

Agentes para manufactura : Se utilizan para mejorar la agilidad y la eficiencia de los procesos industriales de producción. Cada agente es una representación activa de los diferentes componentes del proceso, desde el cliente, a través del representante de ventas, ingenieros, hasta las materias primas.

Agentes para ambientes de diseño : Facilitan la comunicación y cooperación, así como la sincronización de tareas entre varias

personas interesadas en definir problemas de diseño, desarrollando y evaluando posibles soluciones. De igual manera pueden ayudar a fijar juntas de negocios, organizar juntas de trabajo, publicar cartas, y mantener puntos de referencia en la solución de problemas.

Agentes para comercio : La mayoría de las aplicaciones creadas hasta la fecha que pueden agruparse bajo esta denominación, son programas estacionarios capaces de obtener información que se encuentra distribuida a lo largo de una red, analizarla mediante técnicas de inteligencia artificial y generar entonces información útil para el usuario, como tendencias del mercado, especulaciones, ofertas, buenos precios, etc.

Agentes como interfaz máquina - usuario : tienen la finalidad de facilitar la comunicación entre la computadora y el usuario, y hacen uso de características como el reconocimiento de lenguaje natural, presentación de características físicas antropomórficas y de un lenguaje corporal o verbal que muestre incluso estados emocionales.

Agentes para el web : Las aplicaciones para internet basadas en agentes mas importantes son programas que crean y mantienen estadísticas individuales acerca de las preferencias o gustos de las personas que utilizan la red. Ésas estadísticas son elaboradas a partir de calificaciones que el usuario asigna a páginas que el programa le

muestra al azar, y pueden ser utilizadas para encontrar personas con intereses similares, mostrar al usuario páginas de su interés, o enviarle publicidad directa a la medida. Sin embargo, las aplicaciones que se persiguen en éste ámbito son mucho mas complejas que las actuales, e incluyen : comercio electrónico, integración empresarial. bibliotecas digitales, ingeniería concurrente y colaboración a distancia.

En la siguiente sección se muestra cuáles son las características distintivas de la red Internet y qué requerimientos representan éstas en cuanto a las propiedades con las que debe contar un agente que intente desenvolverse en ese medio. De igual manera, establecemos las funciones que el sistema debe realizar y analizamos las características que el grupo de agentes debe tener para lograr la realización de las ésas funciones

2.3. Los Agentes de Software en Internet

2.3.1. El ambiente Internet

Antes de diseñar nuestro sistema multiagentes es necesario conocer las características del medio ambiente en el que sus integrantes se van a desenvolver, por tal motivo, en esta sección se presenta un estudio de las características de la red y del usuario desde el punto de vista del agente individual.

Internet es una fuente de información esencialmente distribuida, es decir, los sitios proveedores de información de interés potencial por parte del usuario se encuentran en lugares geográficamente alejados unos de otros. Por otra parte, es un recurso de naturaleza heterogénea, por la diversidad de los formatos en que la información se presenta y la de los servicios a través de los cuales ésta es accesible. Internet es también un recurso dinámico, debido a que la información que contiene es actualizada continuamente en cuanto a su contenido, localización o simplemente en cuanto a su existencia y disponibilidad en la red.

Internet consta de un conjunto de computadoras interconectadas a través de una estructura de comunicación denominada *subred,* que es la encargada de transferir paquetes de información entre ellas, sin embargo, en la mayoría de los casos sólo es necesario trabajar en el nivel de aplicación, todos los niveles inferiores se encuentran convenientemente abstraídos. Cada computadora tiene la capacidad de recibir peticiones y responder a ellas, así como hacer peticiones y recibir la información desde otra máquina. El intercambio de información entre computadoras funciona con base en una arquitectura cliente-servidor, en la cual sólo las aplicaciones cliente pueden iniciar un intercambio de información mediante la solicitud explícita al servidor. Así mismo, cada computadora tiene la capacidad de ejecutar

aplicaciones de forma local que hagan uso de los recursos que ésa máquina posee.

Cada computadora puede contener un conjunto de documentos accesibles mediante el servicio WWW (documentos HTML, que son los que conciernen a éste trabajo), almacenados en dispositivos de almacenamiento secundario y accesibles por las aplicaciones que se estén ejecutando localmente, con menor o mayor grado de libertad, dependiendo del nivel de seguridad impuesto por el sistema operativo de la computadora.

Una de las principales diferencias de los agentes expuestos en éste trabajo y otros tipos de agentes y aplicaciones convencionales se software, es su movilidad. Un agente móvil puede detener su ejecución en una computadora, viajar a otra a través de la red llevando consigo su estado de ejecución e información adicional, y continuar su ejecución en la nueva máquina.

2.3.2. Los agentes en Internet (trabajos previos).

La mayoría de las aplicaciones de selección y recopilación de información para internet basadas en agentes creadas hasta la fecha (1997), son aplicaciones estacionarias, monolíticas y centralizadas que combinan técnicas de inteligencia artificial con el uso de servicios de

información actualmente disponibles como los buscadores o motores de búsqueda, para tratar de adivinar las preferencias de sus usuarios (no un usuario especifico) mediante estadísticas personales y mediante la comparación de éstas con las estadísticas y gustos de otras personas. Éstas aplicaciones se presentan como un primer intento de personalizar los servicios de búsqueda de información, y se pretende utilizarlas como plataforma para evolucionar a sistemas futuros mas complejos y eficientes [11].

Esta evolución deberá darse a la par con la implantación de una infraestructura que permita la movilidad del código del agente de una forma segura para el proveedor de información y confidencial para el solicitante

En cambio, la solución aquí propuesta se basa en la utilización de un sistema distribuido en donde cada una de las partes integrantes es móvil, y que está dedicado exclusivamente a la satisfacción de una persona. Estos agentes móviles integrantes del sistema deben tener la capacidad de detener su ejecución en una computadora, viajar a otra y continuar ejecutándose allí, y deben llevar con ellos su estado de ejecución y datos adicionales (por ejemplo información obtenida en la computadora visitada).

2.3.3. ¿Qué características deben poseer los agentes para habitar en Internet ?

Los agentes deben tener conocimientos sobre servicios de información (como diccionarios, directorios, buscadores, etc.) disponibles en la red, y tendrán la habilidad para acceder y obtener de ellos información de utilidad

La compatibilidad del software del agente con las plataformas de hardware y software de las computadoras que son visitadas es un requisito muy importante por razones obvias, por lo que es crucial tratar de construir software que siga lineamientos estándar (hasta donde sea posible) con el fin de asegurar la movilidad del código. La utilización de metodologías normalizadas en el diseño y construcción de los agentes también facilita la comunicación entre el agente y el sistema operativo y otros agentes. Aunque actualmente esa normatividad es muy escasa (por no decir que inexistente), si es posible echar mano de lenguajes, plataformas y arquitecturas que han sido propuestas como estándares y han demostrado cierta aceptación general.

Desde el punto de vista de un agente móvil, Internet es una serie de sitios que representan fuentes de información de interés potencial y posibles objetos de visita. La transferencia de un agente de una máquina a otra debe ser completamente segura y debe garantizar la

integridad del programa. En cada sitio visitado el agente hará uso de recursos locales y los compartirá con las aplicaciones locales y otros agentes móviles. Éstos recursos comprenderán procesamiento, espacio en memoria y acceso a información almacenada. De igual manera, el agente tendrá libertad de decidir qué sitios visitar, y la capacidad de moverse de un sitio a otro en el momento que juzgue conveniente.

Un agente debe poder tener acceso (principalmente para lectura) a los documentos almacenados en la máquina, analizar su contenido y compararlo mediante un sistema que le permita decidir si la información que éste contiene puede o no ser útil para el usuario. Finalmente, es necesario que el agente entregue resultados al usuario, comunicándoselos o viajando físicamente hasta su sistema.

2.3.4. Funciones del Sistema.

Podemos establecer los objetivos de alto nivel que el sistema propuesto en este trabajo persigue.

El sistema debe proveer dos modos mínimos de operación : la búsqueda de información a partir de una consulta específica por parte del usuario, y la búsqueda de información por cuenta propia (sin necesidad de una orden) de información que se considere de posible

interés. En el primer caso, el tiempo máximo permitido al sistema para la entrega de resultados queda a disposición del usuario. El resto del tiempo el sistema trabajará en el segundo modo de operación, y entregará los resultados de esas búsquedas libres sólo cuando el usuario se lo permita.

El usuario no debe tratar con cada uno de los agentes de forma individual, sino que el sistema debe presentar un nivel de abstracción tal que todo sea accesible a través de una sola aplicación. Esta aplicación central debe actuar como intermediario entre el sistema multiagentes y el usuario, mostrando los resultados obtenidos por éstos y permitiéndoles actualizar el conocimiento que cada uno de ellos contiene acerca del problema a resolver.

2.3.5. El sistema multiagentes.

Los principales problemas que se presentan en la implementación de nuestro sistema multiagentes son los siguientes [22]:

- ¿Cómo formular, describir, descomponer y asignar el problema y sintetizar resultados entre el grupo de agentes ?
- ¿Cuál será la comunicación y la interacción que existirá entre los agentes ? ¿Qué lenguaje de comunicación y qué protocolo utilizar ? ¿Cómo y cuándo comunicarse ?

- ¿Cómo asegurar que el trabajo conjunto de los agentes lleve a una solución global coherente ?
- ¿Cómo hacer que los agentes individuales razonen acerca de las acciones, planes y conocimiento de otros agentes para que puedan coordinarse ? ¿Cómo razonar acerca del estado del proceso coordinado ?
- ¿Cómo reconocer y reconciliar puntos de vista dispares e intenciones conflictivas entre los miembros de un grupo de agentes ?

En el proceso de diseño de los agentes individuales es necesario considerar :

- La estructura de su conocimiento y la forma como se le dará mantenimiento.
- Cuáles serán sus habilidades de razonamiento.
- Cuáles serán sus habilidades de adaptación y de aprendizaje.
- Cuál será la arquitectura del agente.

Desde el punto de vista del grupo de agentes es necesario considerar los siguientes aspectos :

- La estructura de la organización del grupo. Define la manera en que los agentes comparten dentro del grupo sus creencias, cometidos e intenciones para lograr la meta propuesta.

- La coordinación del grupo. Permite la asignación de recursos escasos entre dos o más agentes, y la comunicación entre ellos de resultados intermedios.
- La cooperación entre agentes. Tiene cuatro objetivos : incrementar la velocidad con que se terminan las tareas mediante el paralelismo, incrementar la gama de problemas solubles compartiendo recursos, incrementar la probabilidad de resolver un problema gracias a la duplicación de la asignación de tareas, y decrementar la interferencia entre tareas evitando las interacciones dañinas
- Las técnicas de negociación. La negociación es el proceso de mejorar el acuerdo entre agentes cuando éstos tienen puntos de vista conflictivos que puedan interferir con el comportamiento cooperativo.
- El comportamiento coherente. El comportamiento coherente se obtiene cuando las acciones de los agentes individuales tienen sentido con respecto a las metas comunes del grupo entero.
- La planeación. Permite que los agentes individuales se hagan cargo de ciertas tareas después de haber examinado cuáles serán las consecuencias de realizar esas tareas en un orden particular.
- El sistema de comunicación. La comunicación permite a los agentes intercambiar información para coordinar sus actividades.

Existe una gran cantidad de sistemas multiagentes desarrollados hasta la fecha, la mayoría de ellos con fines experimentales y de investigación, y dedicados principalmente a probar aspectos teóricos

concernientes a las técnicas de coordinación, cooperación, negociación, planeación, comunicación y comportamiento coherente entre los agentes. De éstos sistemas muy pocos están aplicados al uso mas amigable y eficiente de sistemas de información, y solo unos cuantos han enfrentado el problema de la selección y recopilación información en bases de datos distribuidas.

Algunos de los sistemas que ya hacen uso intensivo de agentes móviles, y que se dedican a la recuperación de información, no aprovechan al máximo las ventajas que ofrece la utilización de código móvil, ya que el agente móvil que viaja a los sitios donde la información reside es un programa único que además de carecer de autonomía, debe satisfacer él solo la variedad completa de necesidades de información del cliente.

2.3.6. Ventajas de la utilización de agentes en Internet

Una de las soluciones actuales más útiles en materia de búsqueda eficiente de información distribuida, son los *buscadores* (search engines) de internet. Conforme la extensión de la red vaya creciendo en el futuro, la eficacia de estas herramientas disminuirá significativamente [13]. El paradigma de utilización de sistemas basados en agentes móviles en internet presenta un conjunto de ventajas que se numeran a continuación :

1. En un motor de búsqueda, la información se busca a partir de palabras clave proporcionadas por la persona solicitante. Esto presupone que el usuario es capaz de formular el conjunto correcto de palabras para traer la información deseada. Consultar con muchas o pocas palabras clave, o incorrectas puede causar la obtención de información irrelevante o la no obtención de información muy importante. En cambio, los agentes son capaces de buscar información de manera mas inteligente, por ejemplo, manejando palabras relacionadas o incluso conceptos e interpretaciones.

2. El mapeo de información se realiza mediante la fabricación de meta - información para cada documento en la red. Éste es un método que consume mucho tiempo y que causa mucho tráfico de datos. Los sistemas que utilizan éste método de selección de información usualmente no cooperan unos con otros, lo que significa que realizan la misma labor de forma redundante. Un agente personal puede crear su propia base de conocimiento acerca de las fuentes de información disponibles en Internet, que se puede actualizar y expandir después de cada búsqueda. Cuando los documentos se mueven a otro sitio, los agentes pueden volver a encontrarlo y actualizar entonces su base de conocimiento. Por otro lado, los agentes pueden comunicarse y cooperar con otros agentes, lo que

permitirá realizar las tareas que tengan asignadas de manera mas rápida y eficiente.

3. Los motores de búsqueda están por naturaleza restringidos al dominio del WWW y otros pocos servicios. Los agentes pueden liberar al usuario humano de tener que preocuparse por detalles sobre las aplicaciones con las cuales encontrar la información.

4. Los motores de búsqueda no siempre pueden ser accesados : el servidor puede estar apagado o puede haber demasiado tráfico en la red. Un agente puede realizar una o mas tareas día y noche, y por su tamaño pequeño puede estar siempre disponible para el usuario. Como buscar información en la red es una tarea que consume demasiado tiempo, tener un agente que se encargue de éste trabajo tiene muchas ventajas, entre ellas podemos mencionar que el agente puede evitar las horas pico en Internet. Un sistema compuesto por múltiples agentes móviles, por otra parte, explota el paralelismo, ya que cada uno de ellos se ejecuta en una máquina diferente, y por lo tanto conduce a una solución mas rápida.

5. Los buscadores de Internet son independientes del dominio en la forma en que tratan la información que manejan. Los términos de mayor ocurrencia en los documentos disponibles se extraen fuera de su contexto y se guardan como palabras clave individuales. Los

agentes de software, por otro lado pueden buscar la información basándose en contextos.

6. La información en Internet es muy dinámica, por lo que frecuentemente los buscadores hacen referencias a documentos que ya no existen o se han movido, y no aprenden de esas búsquedas fallidas. Además, el usuario no puede recibir información que actualice sus recursos conocidos. Los agentes, por otro lado, pueden ajustarse a sí mismos de acuerdo a las preferencias, gustos y deseos de usuarios individuales, aprendiendo de la forma en que realizó anteriormente sus tareas y la forma en que su dueño reaccionó ante el producto. Los agentes también pueden buscar continuamente en la red documentos nuevos que puedan satisfacer las necesidades del dueño, e incluso sugerirle nuevos tipos de documentos.

Un sistema multiagentes utilizado para la recopilación de información distribuida presenta además las siguientes ventajas [27]:

- Múltiples agentes ofrecen concurrencia, que representa una gran ventaja en situaciones en que el tiempo de respuesta es crítico, o cuando el espacio de búsqueda es muy grande, como en el caso de una red.
- Las tareas que es necesario realizar para satisfacer una consulta explícita puede ser frecuentemente descompuesta en subtareas

relativamente independientes con pocas interdependencias. Un agente dedicado a una de esas subtareas puede funcionar de manera autónoma, pero necesita cierta coordinación previa para considerar posibles interdependencias.

- Cuando un sistema maneja enormes cantidades de datos, la computación distribuida en los sitios donde la información reside puede ser una solución mejor comparada con migrar datos a un sitio centralizado de proceso. En lugar de reunir datos dispersados por servidores de información en red, en un lugar centralizado, y entonces lograr una respuesta coherente a una consulta, los agentes pueden residir en las fuentes y transportar solamente un subconjunto mucho menor de datos al sistema central para su posterior procesamiento.
- Las arquitecturas basadas en agentes ofrecen modularidad, robustez, y separación de tareas. Los agentes pueden ser construidos y mantenidos separadamente, y un agente puede trabajar sobre otro, de manera que se incremente la abstracción de las fuentes de información.

2.4 Plataformas Disponibles para Desarrollo e Implementación.

Recientemente han comenzado a aparecer diversas plataformas comerciales y no comerciales que permiten la creación de agentes móviles, que tienen como base lenguajes conocidos de propósito general, o que hacen uso de un lenguaje propio. El objetivo de ésta sección es realizar un breve análisis de las características que cada una de ellas ofrece, y hacer un estudio comparativo para la selección de la mas apropiada.

El lenguaje de programación o plataforma utilizados definen la infraestructura de la operación y funcionalidad de los agentes.

2.4.1. Características de la infraestructura a considerar.

Facilidad de ejecución. Esta es una variable dependiente de la compatibilidad del hardware con el software necesario para ejecutar los programas del agente y los que proveen el ambiente del agente. El lenguaje de programación para software de agentes determina el rango de funcionalidad posible en una aplicación de agentes. Hasta el momento, ésta es un área de intenso estudio e investigación por parte de instituciones académicas y comerciales. Dado que las aplicaciones de agentes difieren enormemente, y dado que el campo de ésta

tecnología aún esta evolucionando, los expertos rara vez se ponen de acuerdo sobre el lenguaje de programación apropiado para el desarrollo de aplicaciones con agentes. Generalmente, las opciones caen dentro de tres categorías: lenguajes de programación general y lenguajes para scripts, lenguajes de código móvil de propósito general, y finalmente lenguajes de código móvil escritos específicamente para aplicaciones de agentes.

a) Lenguajes de programación de propósito general y lenguajes para scripts.

Los lenguajes de programación de propósito general favorecidos para escritura de aplicaciones de agentes debido a su eficiencia incluyen a los lenguajes C y C++, y a el lenguaje Lisp, el cual es favorecido por su flexible estructura simbólica. Lisp es comúnmente utilizado en aplicaciones de inteligencia artificial. Muchas de las aplicaciones sencillas, las cuales son consideradas aplicaciones inteligentes, son en su mayor parte escritas con lenguajes para el desarrollo de scripts (guías) como Structured Query Language (SQL), comúnmente utilizado para consultas a bases de datos, o como Perl, comúnmente utilizado para el desarrollo de scripts de comandos para el WWW.

b) Lenguajes de código móvil de propósito general.

El contar con un lenguaje que produce código móvil añade al propio código la flexibilidad necesaria para ser transmitido a través de la red y ejecutado en otro punto de ésta. Actualmente existen diversos sistemas de código móvil. Java es considerado un lenguaje similar a C++, pero escrito con características extras de movilidad y seguridad para su código.

Safe-Tcl, un poco más antiguo que Java, es otro de los lenguajes de código móvil pero con menor rendimiento. El objetivo de Safe-Tcl es proporcionar un lenguaje para aplicaciones basadas en correo electrónico. Otros lenguajes de código móvil, cada uno con sus seguidores devotos, son Python, scheme48, Guile, Obliq, Logicware, ScriptX, y Phantom.

c) Lenguajes de código móvil para aplicaciones de agentes inteligentes.

Aunque los lenguajes de código móvil mencionados proveen la funcionalidad necesaria para desarrollar aplicaciones con agentes inteligentes móviles, no fueron desarrollados especificamente para esta clase de aplicaciones. Algunas instituciones académicas y comerciales han desarrollado lenguajes de programación de agentes diseñados para explotar las aplicaciones de los agentes inteligentes móviles.

El profesor Yoav Shoham de la Universidad de Stanford ha propuesto un "nuevo paradigma, basado en la visión social de la computación". Su concepto de Programación Orientada a Agentes (AOP) ha conducido al desarrollo de AGENT0, un lenguaje de programación para agentes inteligentes. Este lenguaje a la fecha no esta completamente desarrollado pero esta disponible en Internet.

IBM se encuentra en el proceso de modificar su exitoso lenguaje de programación REXX, bien conocido por su facilidad de uso y sus capacidades en el desarrollo de prototipos, para destinarlo a la tecnología emergente del cómputo basado en agentes. IBM espera que el nuevo lenguaje Object-REXX llegue a ser el estándar para la programación de agentes.

Facilidad de comunicación. Esencialmente la facilidad de comunicación tiene que ver con el intercambio de información entre diversos agentes. La facilidad de comunicación debe soportar comunicación síncrona y asíncrona, y debe ser capaz de comunicar simultáneamente a múltiples agentes diferentes.

La Arquitectura de Agente Unificada, propuesta por Marc Belgrave en la Universidad McGill en Montreal, Canadá, define la facilidad de comunicación como un conjunto de protocolos estándares que permiten que la comunicación se lleve a cabo. A la fecha ninguna facilidad de

comunicación estándar ha emergido, sin embargo el ARPA Knowledge Sharing Effort (KSE) esta intentando desarrollar técnicas y una metodología para construir bases de conocimiento en gran escala que facilite la comunicación entre agentes. El principal resultado es el KQML (Knowledge Query and Manipulation Language) un formateador de mensajes y protocolo para el manejo de mensajes que soporta el intercambio de conocimientos entre agentes al tiempo de ejecución. KQML puede ser usado como un lenguaje para un programa de aplicación para interactuar con un sistema inteligente o para que dos o más sistemas inteligentes compartan el conocimiento para resolver problemas en forma cooperativa.

Facilidad de transporte. Mientras que la facilidad de comunicación tiene que ver con lo que el agente está comunicando, la facilidad de transporte tiene que ver con la forma en que el agente se está comunicando. La facilidad de transporte permite el movimiento de un agente desde un ambiente de ejecución a otro para llevar a cabo una tarea. También permite la distribución de agentes no inicializados (los agentes inicializados son los que comienzan ejecutar su tarea dentro de la máquina cliente, y requieren continuarla en la máquina servidor, los no inicializados son aquellos que comienzan su tarea en la máquina servidor). La facilidad de transporte generalmente soporta protocolos de transmisión de datos establecidos tales como HTTP, SMTP, y TCP/IP.

Facilidad para empacar. Esto provee un método estándar para empacar agentes junto con su información asociada. Sin tomar en cuenta la estructura interna, todos los agentes deben encapsular el estado, la autenticidad, y los objetivos de la información así como sus capacidades, y la metodología usada en la planeación del manejo de la información. A la fecha ninguna facilidad estándar para empacar ha emergido.

Seguridad integrada. La identificación y "huellas digitales" de los agentes es un atributo intrínseco de la infraestructura de la arquitectura de un agente. La seguridad es un tema de infraestructura amplio que aplica a todos los componentes previamente discutidos.

La infraestructura del agente debe proveer un método seguro inherente que permita determinar el propietario y el lugar de origen del agente. En la mayoría de las aplicaciones, cada agente posee un "pasaporte" el cual cifrara esta información con el objetivo de que no sea alterada y pueda ocasionar posibles daños.

Los métodos de seguridad utilizados en la infraestructura de un agente deben considerar también la protección de recursos (no abusar de las computadoras o redes), privacidad de la información, y procedimientos y reglas de terminación del agente. Generalmente,

cada aplicación de agentes, utiliza sus propios protocolos de seguridad para manejar los requerimientos específicos de la aplicación.

2.4.2. Características Deseables del Sistema Producido

Además de las características dignas de consideración mencionadas anteriormente, es deseable contar con agentes que cumplan las siguientes :

Sencillos en su implementación. Cada agente debe realizar alguna tarea sencilla, de tal forma que cada uno de los agentes en si mismo debe ser simple en su representación interna.

Producir poco overhead. Como los agentes deben ser sencillos y pequeños no deben implicar una gran carga para la máquina los hospeda (en recursos de procesamiento y memoria), así como para la red que los transporta.

Flexible. Como cada agente realiza una tarea específica y esta tarea debe ser lo mas pequeña posible, es relativamente sencillo sustituir algún agente para que realice alguna otra tarea.

Escalable. Solo es necesario incluir otros agentes cuando se realicen otras funciones diferentes para que el sistema sea capaz de realizar nuevas tareas

2.4.3. Lenguajes apropiados

Teniendo en cuenta los requerimientos mencionados antes, analizaremos a continuación las características de los lenguajes cuyo grado de desarrollo y aceptación permite confiar en que serán una pieza clave en la evolución futura de las aplicaciones basadas en agentes móviles.

2.4.3.1. Telescript

Hasta ahora, el lenguaje de programación de agentes que ha recibido la mayor atención es Telescript de General Magic. Éste es un lenguaje de programación remota completamente orientado a objetos y con una sintaxis muy parecida a Smalltalk, una plataforma que permite la creación de aplicaciones de red activas y distribuidas. Telescript permite a los programadores escribir aplicaciones para facilitar a los agentes la navegación a través de una variedad de servicios y de diferentes sistemas de cómputo para ejecutar tareas para usuarios. Además de proveer el lenguaje de programación para la ejecución del ambiente, también cubre aspectos de comunicación y seguridad. Su

tecnología es considerada superior a la de sus competidores para aplicaciones de agentes, aunque la tecnología no es tan madura como muchos de los lenguajes de propósito general y lenguajes de código móvil. Telescript se encuentra bien situado en aplicaciones de comercio electrónico.

Telescript implementa los siguientes conceptos principales : lugares, agentes, viajes, reuniones, conexiones, autoridades, y permisos.

Lugares. Telescript modela una red de computadoras como una colección de lugares. Cada lugar ofrece un servicio a un agente móvil que entra en él. Los servidores constituyen algunos lugares, y otros las computadoras cliente.

Agentes. Telescript modela una aplicación de comunicaciones como una colección de agentes. Cada agente ocupa un lugar particular. Sin embargo, un agente puede moverse de un lugar a otro. Cada lugar es representado y ocupado permanentemente por un agente estacionario que provee el servicio.

Después de que General Magic reconociera que la adopción generalizada de Java evitaría la aceptación de su lenguaje Telescript, éste y todos los productos basados en él, quedaron discontinuados. A pesar de que la compañía se apoyaba grandemente en la funcionalidad

específica que proveía Telescript en el desarrollo de agentes móviles, trabaja ahora en encontrar maneras de proveer esa funcionalidad mediante código cien por ciento Java.

Actualmente, siguiendo esa línea de investigación, se encuentra desarrollando Odyssey, un sistema de agentes implementado como un conjunto de librerías de clases Java, que brinda el soporte para desarrollar aplicaciones móviles y distribuidas.

Java es un lenguaje de programación orientado a objetos desarrollado por Sun Microsystems en 1991. Modelado a partir de C++, Java fue diseñado para ser pequeño, simple, y portable entre plataformas y sistemas operativos diversos, tanto a nivel fuente como a nivel binario.

2.4.3.2. Java

Las principales características que han hecho de Java un lenguaje altamente utilizado, y que le han conferido su fama característica se numeran a continuación [5]:

- Independencia de plataforma

  Es una de las ventajas mas significativas que tiene Java sobre otros lenguajes de programación, particularmente para sistemas que necesitan trabajar en plataformas diversas.

- Es orientado a objetos

  Permite crear programas flexibles y modulares, así como la reutilización de código. Contiene una librería de clases que provee tipos básicos de datos, capacidad de entrada y salida, y otras funciones de utilidad. Estas clases soportan también el desarrollo de aplicaciones que utilizan la red, los protocolos de internet y funciones para la creación de interfaces gráficas de usuario.

- Es fácil de aprender

  Una de las metas de diseño de Java es que sea fácil de escribir, de compilar, de analizar y de aprender. Java es aún mas fácil de aprender si se sabe C++, ya que fue diseñado a partir de este lenguaje, pero sin muchas de sus complicaciones.

Las dos herramientas mas útiles del Java Development Kit (JDK) son el compilador y el intérprete. A diferencia de los compiladores de otros

lenguajes, el compilador Java (javac.exe) no genera código ejecutable, que por definición debe ser específico para una máquina y sistema operativo, sino "bytecode".

El bytecode es código especialmente diseñado para ser procesado por una máquina virtual en lugar de un microprocesador. Esta máquina virtual es parte de un programa intérprete (java.exe), producido específicamente para cierta plataforma y sistema operativo.

El compilador Java utiliza un archivo .java de tipo texto con el código fuente, y genera un archivo .class que contiene el bytecode, mismo que puede ser transportado y ejecutado en cualquier otro sistema para cuya plataforma de hardware y de software se encuentre disponible el JDK, específicamente, el intérprete Java.

Las ventajas arriba mencionadas, aunadas a la alta disponibilidad del lenguaje y de las plataformas de desarrollo de agentes basadas en él, hacen de Java el lenguaje mas apropiado para la creación del sistema que nos ocupa.

2.4.4. Las plataformas basadas en Java.

En la actualidad existen varias arquitecturas de agentes implementadas en diversos lenguajes tales como Telescript, TCL/TK y Java.

Estos lenguajes son relativamente nuevos y han mostrado su facilidad de uso sobre Internet en diversas plataformas. Telescript se ha distinguido por ser un lenguaje dirigido explícitamente para la creación, manejo, traslado y monitoreo de agentes, no obstante este lenguaje no es de dominio público y por lo tanto su uso ha sido restringido a proyectos especiales de compañías tales como AT&T y General Magic, dado que tiene un costo elevado. TCL/TK es un lenguaje sencillo y de fácil uso para el desarrollo de prototipos y aplicaciones académicas, pero su principal problema es su baja velocidad de ejecución de las aplicaciones de usuario. Por lo anterior, en este trabajo se da preferencia a la utilización de frameworks basados en Java, por ser ampliamente disponible sobre Internet, por su portabilidad y su costo inexistente

Todas las plataformas comerciales basadas en Java disponibles actualmente, comparten ciertas características que surgen del propio lenguaje [10]:

- *Proveen un servidor de agentes de algún tipo.* El servidor de agentes es el punto de contacto en una máquina determinada, la caja de arena en la que los agentes se mueven y actúan.
- *Los agentes pueden migrar de un servidor a otro, llevando consigo su estado de ejecución.* Algunos sistemas migran los agentes automáticamente de acuerdo a un itinerario, mientras que otros permiten a cada agente determinar sus propios destinos.
- *Los agentes pueden cargar su código de una variedad de fuentes.* En general, dado que todos los sistemas usan una versión especializada del cargador de clases de Java, pueden cargar clases Java del sistema local de archivos, el web, y servidores ftp.
- *Todos son 100 por ciento Java y usan características del JDK 1.1.* Esto significa que deben corren en cualquier computadora con un runtime Java 1.1.

Antes de describir los frameworks vale la pena mencionar dos de las extensiones que actualmente ofrece java para aplicaciones de cómputo distribuido, y las cuales brindan al software de agentes su funcionalidad y movilidad característica: Object Serialization [1] y Remote Method Invocation [2]. Mas adelante analizamos varias plataformas de desarrollo de agentes que hacen uso de Java como lenguaje base.

2.4.4.1. Object Serialization

Es un conjunto de clases que extienden las clases centrales de entrada/salida de Java para que soporten la codificación de objetos y los objetos alcanzables desde ellos en la forma de un flujo de bytes, y además soporta la reconstrucción complementaria de los mismos a partir de ese flujo de bytes [1]. Object Serialization es usado para proporcionar un cierto grado de persistencia y para la comunicación por medio de sockets o RMI (Remote Method Invocation). La codificación por default de los objetos protege datos privados y transitorios. Una clase puede implementar su propia codificación externa y es entonces responsable únicamente del formato externo. Object Serialization viene ahora incluido como parte del JDK (Java Development Kit) versión 1.1 y posteriores.

2.4.4.2. RMI (Remote Method Invocation)

Los sistemas distribuidos requieren que el procesamiento se lleve a cabo en diferentes espacios de direccionamiento, lo que puede significar incluso diferentes máquinas [2]. Además, es necesario que los procesos sean capaces de comunicarse entre sí. Los sockets son un medio de comunicación flexible y suficiente para comunicarse en general, sin embargo, requieren que el cliente y el servidor trabajen con protocolos de nivel de aplicación para codificar y descodificar los

mensajes, y tales protocolos son difíciles de utilizar y su diseño es complicado. RPC (Remote Procedure Calls) provee una abstracción conveniente de los sockets, mediante llamadas a procedimientos remotos que parecen ser locales, sin embargo, no funciona adecuadamente en sistemas de objetos distribuidos, donde es necesaria la comunicación entre objetos a nivel de aplicación residiendo en diferentes espacios de direccionamiento. Con el objeto de mantener la semántica de invocación de métodos remotos, los sistemas de objetos distribuidos requieren la utilización de RMI (Remote Method Invocation). En tales sistemas, un objeto substituto local (llamado "stub") maneja la invocación de un objeto remoto.

RMI permite crear aplicaciones distribuidas, en las cuales los métodos de objetos remotos pueden ser invocados desde otras máquinas virtuales Java. Un programa Java puede hacer una llamada a un objeto remoto una vez que este obtiene una referencia al objeto por medio de buscar el objeto remoto en el servicio de nombres provisto por el RMI, o al recibir la referencia como un argumento. Por otra parte un cliente puede llamar a un objeto remoto en un servidor, y este servidor puede ser un cliente de otros objetos remotos.

Características del sistema RMI de Java.

- Soporta la invocación remota de objetos en diferentes máquinas virtuales.
- Integra el modelo de objetos distribuidos dentro del lenguaje Java en una forma natural
- Mantiene la semántica de los objetos.
- Preserva el mecanismo de seguridad del ambiente de ejecución de Java
- Recolección distribuida de basura para la eliminación de objetos remotos.
- Activación de objetos persistentes para atender peticiones.

A continuación cuatro de las principales implementaciones de agentes utilizando Java son mostradas. A saber : Java Agent Template, Aglets Workbench, Odyssey, y Voyager.

2.4.4.3. JAT (Java Agent Template)

El JAT provee un completo template funcional escrito en su totalidad en Java para la construcción de software de agentes, los cuales se comunican uno a uno con una comunidad de otros agentes distribuidos a través de Internet [9]. Aunque parte del código que define a un agente es portable, los agentes creados con JAT no son migratorios, mas bien

estos tienen una existencia estática sobre una máquina. Actualmente, todos los mensajes de agentes usan KQML como un protocolo de alto nivel en el intercambio de mensajes. El JAT incluye funcionalidad para el intercambio dinámico de "recursos", lo cual puede incluir clases de java, archivos de datos e información en línea dentro de los mensajes KQML.

JAT puede ser ejecutado como una aplicación standalone o como un applet a través del appletviewer. La coordinación entre los agentes es llevada a cabo por medio de un servidor de nombres de agentes. La arquitectura del JAT fue especialmente diseñada para permitir el reemplazo y especialización de componentes que agreguen funcionalidad, tales como GUI's, mensajes de bajo nivel, interpretación de mensajes y manejo de recursos. Consecuentemente, el JAT puede ser usado como una plataforma para la construcción de un amplio tipo de agentes para diferentes aplicaciones.

Todos los agentes JAT mantienen tanto clases como datos usando objetos, los cuales con subclases de la clase JavaAgent.resource.Resource. Los recursos de clase incluyen lenguajes (principalmente manejadores de protocolo que permiten parsear un mensaje y proveen algunas semánticas de alto nivel) e intérpretes (esencialmente manejadores de contenido, provee una especificación procedural de como un mensaje, construido acorde a

una ontología especifica, debería ser interpretado). Se asume que la implementación de cada clase Interpreter esta basada en alguna especificación formal (ontology) de las semánticas de mensaje. Una propiedad del JAT da la posibilidad de que estos recursos puedan ser dinámicamente intercambiados entre agentes de una forma "just-in-time" (una subclase de Resource, RetrievalResource es utilizada para este propósito). Esto permite a una agente procesar correctamente los mensajes, cuyo lenguaje e interprete son desconocidos, al adquirir las clases e interprete de forma dinámica.

2.4.4.4. Aglets Workbench de IBM

Características Generales

Aglets Workbench (AWB) es un ambiente para la construcción de aplicaciones basadas en red que usan agentes móviles para la búsqueda, acceso y manipulación de datos [6]. Según la arquitectura de este framework los agentes pueden ser enviados desde cualquier computadora y transportados a otra para continuar su ejecución. Al llegar a la máquina anfitriona, los agentes presentan sus credenciales y obtienen acceso a los servicios y datos locales. La computadora remota puede también servir como un intermediario al agrupar agentes con intereses similares y metas compatibles, brindando de esta forma un lugar de reunión en el cual los agentes puedan interactuar.

El Aglet Java extiende el modelo de código móvil a través de la red utilizado por los applets. Como un applet, las clases que constituyen el aglet pueden migrar a través de la red. Pero al contrario de éstos, cuando un aglet migra, lleva consigo su estado de ejecución.

Un applet es un programa que se desplaza de un servidor a un cliente. Un aglet puede moverse de un host a otro dentro de la red. Además, como un aglet carga su estado de ejecución a donde quiera que va, puede viajar secuencialmente a muchos destinos en la red, incluyendo regresar eventualmente a su host original.

Un aglet Java es similar a un applet en que corre como un hilo (o múltiples hilos) dentro de el contexto de una aplicación Java host. Ésa aplicación instala un administrador de seguridad para reforzar las restricciones en las actividades de un applet. Para pedir los archivos de las clases del applet, la aplicación crea cargadores de clases (class loaders) cuya función es requisar los archivos de clases de un servidor HTTP.

De igual manera, un aglet requiere una aplicación Java host (un aglet host), para correr en una computadora. Cuando los aglets viajan a través de la red, migran de un aglet host a otro. Cada aglet host instala un administrador de seguridad para restringir las actividades de los

aglets a los que no se tiene confianza. Los host transportan los aglets mediante cargadores de clases cuya función es cargar los archivos de clases y estado de ejecución de los aglets desde un host remoto.

AWB soporta el desarrollo de agentes es a través del uso de patrones. Esta plataforma tiene un numero de patrones de uso de agentes de alto nivel. Estos patrones de uso describen relaciones comunes entre agentes tales como Master-Slave (maestro-esclavo), Messenger-Receiver (mensajero-receptor), y Notifier-Notification (notificador-notificado). Estos patrones son presentados como un conjunto de clases que pueden ser usados como esqueletos por el desarrollador

Las metas de diseño de Aglets Workbench son: simplicidad y extensibilidad, independencia de plataformas, seguridad (para el host). Se aspira a convertir ésta plataforma en un estándar de la industria. Está completamente documentado y permite fácilmente la creación de agentes móviles para sistemas de información.

Características Específicas :

Algunas de principales características que tiene el framework son:

Acceso a datos corporativos. acceso a bases de datos corporativas es esencial en muchas de las aplicaciones de agentes móviles. Aglets Workbench cuenta con varios packages para acceso de datos, incluyendo JDBC/DB2 y JoDax. Estos packages evitan que el desarrollador de agentes tenga que hacer uso de métodos primitivos para accesar a bases de datos sin importar si son locales o remotas.

ATP (Agent Transfer Protocol). El ATP (protocolo para transferencia de agentes) es usado para transferir agentes sobre la red. ATP es un protocolo a nivel de aplicación para sistemas de información distribuidos basados en agentes. Este protocolo esta basado en internet y el uso de URL's para la localización de recursos de agentes, ATP ofrece un protocolo uniforme y transparente, independiente de la plataforma que se este utilizando para la transferencia de agentes entre computadoras.

Mientras que los agentes móviles pueden ser escritos en muchos diferentes lenguajes y por una variedad de sistemas de agentes específicos del proveedor, ATP ofrece la oportunidad para manejar la movilidad en una forma uniforme y general.

La primera versión de ATP que es la que se utiliza actualmente, esta enteramente escrito en Java e implementada como un package completamente independiente y documentado dentro del Aglets

Workbench. El ATP esta formado por un conjunto de clases altamente portables que proveen un API para crear daemons, conectarse a sitios ATP, y generar solicitudes y respuestas ATP

Administrador de Agente Visual. Tahiti es un manejador de aglets visual basado en el Aglets Framework. Tahiti usa una interfaz de usuario gráfica para monitorear y controlar la ejecución de los aglets en la computadora local. A través de una interfaz drag and drop uno puede hacer que dos aglets se comuniquen entre si, o mandar a un aglet a un sitio en particular. Tahiti además de ser una herramienta de administración del sistema; es una herramienta de escritorio para que los usuarios de los aglets manipulen a sus agentes de la misma forma en que usan un browser de web como herramienta fundamental.

Seguridad. La seguridad es un punto importante para los usuarios de agentes móviles. Por un lado los agentes pueden ser usados para un gran cantidad de actividades en beneficio del usuario, no obstante, por otra parte ellos podrían llegar a ser una tierra fértil para la creación de virus. Recibir agentes desconocidos de la red es potencialmente una invitación abierta para una gran cantidad de problemas.

El Aglet Framework soporta un modelo de seguridad por capas. La primera capa es el sistema del lenguaje mismo. Los fragmentos de código importados en los agentes son sujetos a una serie de pruebas

para asegurar que el formato del código que transportan es correcto, y finaliza con una serie de revisiones realizadas por el verificador de bytecode de Java.

En la siguiente capa esta el administrador de seguridad, el cual permite a los usuarios del Aglets Framework implementar sus propios mecanismos de seguridad. Tahiti implementa un administrador configurable que provee un alto grado de seguridad para la computadora huésped y su propietario. La configuración por default de seguridad es muy restrictiva. Cualquier intento por parte de un agente para accesar un archivo para el cual el acceso no ha sido proporcionado será considerado como una violación de seguridad, y al agente no se le permitirá el acceso.

En la tercera y ultima capa esta el API de seguridad de Java, el cual es un framework sencillo para el desarrollador de agentes. Este incluye cierta funcionalidad de seguridad en sus agentes. Dicha funcionalidad incluye a su vez el uso de técnicas de criptografía, firmas digitales, cifrado, y autentificación.

Estandarización. Tanto el ATP como el Aglets Workbench han sido propuestos a la OMG (Object Managment Group) Grupo de Administración de Objetos, para su aceptación como el estándar para agentes móviles.

Aún mas, la especificación del ATP es de dominio publico. El protocolo ha sido modelado sobre HTTP, y propone una forma estándar de transportar agentes independientemente de cualquier implementación particular. Actualmente Aglets Workbench puede ser obtenido gratuitamente desde Japón.

2.4.4.5. Voyager de ObjectSpace

ObjectSpace es una empresa que hasta hace unos meses no tenía nada que ver con la tecnología de agentes móviles. De hecho, la primera liberación de Voyager 1.0 (beta 2.1) fue realizada en Junio de 1997. Voyager está disponible sin costo para su evaluación, con restricciones para las aplicaciones de uso comercial [7].

El paquete contiene solamente los archivos en código fuente en Java, y el desarrollador debe construirlos. La razón de esta decisión es el uso del Objeto Virtual, la clave de la infraestructura de control y comunicación interagentes de Voyager. El Objeto Virtual es una especie de proxy post-procesado de un objeto remoto o agente. Otros sistemas que utilizan un mecanismo como el RPC (por ejemplo RMI), requieren que el desarrollo pase por una serie de pasos para describir primero la interfaz y después la implementación de un objeto. Voyager toma cualquier clase Java (código fuente o archivo de clase) y lo

modifica con la herramienta Virtual Code Compiler, que crea el Objeto Virtual de la clase fuente. Este Objeto Virtual permite que los objetos sean manipulados de forma remota.

La facilidad de comunicación en Voyager es muy flexible. Permite la comunicación síncrona, asíncrona y futura. De igual manera, la forma de migración de los agentes es innovadora y tiene algunas ventajas sobre sus competidores, principalmente porque no es necesario correr un servidor de agentes en todos los sitios de la red que el agente visitará. Esto se debe a que un objeto virtual puede migrar no solo entre servidores de agentes, sino también entre máquinas virtuales Java de otros objetos virtuales arbitrarios.

La principal desventaja de los objetos virtuales es que cambian la manera en que el desarrollador construye software. Un archivo Java puede ser procesado por la herramienta VCC solamente si el archivo es sintácticamente correcto. Muy frecuentemente, sin embargo, ese archivo hará referencia a otras clases de objetos virtuales. La maraña de interdependencias que resulta puede ser muy frustrante.

2.4.4.6. Odyssey de General Magic

Con el advenimiento de Java y sus promesas de independencia de plataforma, General Magic comenzó a desarrollar Odyssey, un sistema

de agentes implementado completamente en Java, que incorpora algunos de los conceptos previamente desarrollados para Telescript. El paquete Odyssey [8], al igual que Aglets Workbench, viene con clases precompiladas que permiten una rápida ejecución del sistema y las aplicaciones de demostración. Además, debe hacerse correr un proceso "rmiregistry" (proveído por Sun como parte del JDK) en cada máquina en el que el servidor de agentes se ejecutará.

Una característica única de Odyssey es que los agentes pueden usar una clase distribuida con Odyssey para enviar su estado de ejecución y otra información, a una salida estándar, una ventana, o un archivo en el host donde se encuentra corriendo, lo cual facilita el monitoreo de su ejecución y la búsqueda de errores de programación. Sin embargo, por el hecho de que Java no soporta el despliegue remoto de información, así como que los archivos no son serializables, este sistema presenta problemas cuando el agente no se encuentra en la máquina virtual que lo originó.

General Magic aún no ha determinado si Odyssey se convertirá en un producto comercial, aunque han prohibido específicamente el desarrollo comercial de aplicaciones basadas en Odyssey.

En conclusión, podemos decir que el AgentTemplate toolkit es una excelente herramienta para la construcción de agentes que requieren el

uso de inteligencia artificial. Bajo el AgentTemplate, las interacciones entre agentes están basadas sobre Knowledge Query and Manipulation Language (KQML), un lenguaje diseñado específicamente para aplicaciones de inteligencia artificial. Mientras esta es una solución sofisticada para algunas de las aplicaciones basadas en agentes, los agentes por si mismos son estacionarios y no son apropiados para verdaderas aplicaciones móviles. Además, el uso de un lenguaje por separado para el intercambio de información es una complejidad que nuestra aplicación no requiere.

Por otra parte el Aglets Workbench proporciona algunas características que considero fundamentales para nuestro sistema:

Objetos móviles y estacionarios.
Un método para el manejo de objetos, mensajes y datos
Objetos autónomos y pasivos
Procesamiento síncrono y asíncrono
Operaciones con y sin conexión
Ejecución secuencial y en paralelo

Esta capacidad de movilidad resulta muy conveniente para su uso en el WWW.

Existen varias razones por las que Aglets Workbench ha ganado un terreno mucho mayor entre los programadores de agentes en comparación con las otras tecnologías mencionadas. La principal de ellas es su limpio diseño. El Aglets Workbench es tan maduro como para imitar varios modelos tradicionales de Java : los applets, los métodos callback del AWT (Abstract Window Toolkit), Java Beans, y un sistema de mensajería por publicación y subscripción que es similar al emergente Java Messaging Service (JMS). Estas características y todas las mencionadas anteriormente, permiten elegir a Aglets Workbench como la plataforma mas adecuada para la construcción de nuestro sistema multiagentes.

Parte III Implementación del Sistema Multiagentes

3.1. Aglets Workbench

3.1.1. Elementos del Aglet API

J-AAPI (Java Aglets Aplication Programming Interface) es un estándar propuesto que sirve como interface entre los aglets y su ambiente [3][6]. Contiene métodos para inicializar un agente, para manejo de mensajes, y para enviar, retraer, activar, desactivar, clonar y liberar un aglet.

Las interfaces y clases del J-AAPI son las siguientes :

Interfaces : AgletContext
FutureReply
MessageManager

Clases : Aglet
AgletIdentifier
AgletProxy
Message

La finalidad de cada una de estas clases e interfaces se explica a continuación.

3.1.2. El modelo de objetos de Aglets Workbench

El modelo de objetos de los aglets define un conjunto de abstracciones del comportamiento que se necesita para implantar la tecnología de agentes móviles en redes de área local y amplia de tipo Internet [3]. Las abstracciones clave que encontramos en este modelo son : aglet, context (contexto), proxy (apoderado), mensaje, e identificador.

Un aglet es un objeto móvil programado en Java, que visita hosts habilitados para hospedarlo en una red de computadoras. Tiene la capacidad de ejecutarse de manera autónoma y de responder a los mensajes que recibe.

Un contexto es el lugar de trabajo de un aglet. Es un objeto estacionario que provee un medio para administrar y mantener los aglets en ejecución de un ambiente uniforme donde el sistema host queda protegido contra aglets maliciosos.

Un proxy o apoderado es el representante de un aglet. Sirve como un escudo que protege al aglet del acceso directo a sus métodos públicos. El proxy también provee transparencia de locación al aglet, es decir, esconde la locación real del aglet.

Un mensaje es un objeto intercambiado entre aglets. Permite intercambio de mensajes en forma síncrona y asíncrona. El intercambio de mensajes puede ser utilizado por los aglets para colaborar e

intercambiar información de una manera muy flexible. Se dispone además de un administrador de mensajes que permite el control de concurrencia de los mensajes entrantes.

Un identificador es asignado a cada aglet. Este identificador es globalmente único e inmutable a lo largo del ciclo de vida del aglet.

El comportamiento soportado en el modelo de objetos del aglet incluye : creación, clonación, despacho, retractación, desactivación, liberación, y paso de mensajes.

La creación de un agente tiene lugar en un contexto. Al nuevo aglet se le asigna un identificador insertado en el contexto, y es inicializado. El aglet comienza su ejecución tan pronto como sea inicializado con éxito.

La clonación de un aglet produce un aglet copia del original casi idéntico en el mismo contexto. Las únicas diferencias son el identificador asignado y que la ejecución del clon comienza desde el principio. Hay que notar que los hilos de ejecución no son clonados.

Al despachar un aglet de un contexto a otro, éste será removido de su contexto original e insertado en el contexto destino, donde volverá a comenzar su ejecución.

La retractación de un aglet lo removerá de su contexto actual y lo insertará en el contexto desde el cual la retractación fue solicitada.

La desactivación de un aglet es la habilidad para removerlo temporalmente de su contexto actual y guardarlo en un medio de almacenamiento secundario. La activación de un aglet lo restaurará en su contexto actual.

La liberación de un aglet detendrá su ejecución actual y lo removerá de su contexto actual.

El paso de mensajes entre los aglets involucra enviar, recibir y manejar mensajes de manera tanto síncrona como asíncrona.

En materia de seguridad, es necesario asegurar que ningún aglet comprometa la seguridad del host. Los aglets pueden ser clasificados según la confianza que se les tenga. El administrador de seguridad determina si a un aglet se le permite accesar el sistema de archivos, la red y a otros aglets. La decisión de confiar en un aglet depende enteramente del host.

3.1.3. Descripción de las clases e interfaces centrales.

La Clase Aglet

La clase Aglet es la clase clave en el J-AAPI. Esta es la clase abstracta que se usa como base para crear aglets propios. La clase Aglet define métodos para controlar su propio ciclo de vida, así como métodos que se deben sobreescribir en las subclases por el programador del aglet, y le provee las herramientas necesarias para adecuar su comportamiento. Estos métodos son invocados sistemáticamente por el sistema cuando ocurren ciertos eventos en la vida del agente. La tabla a continuación muestra la relación entre esos eventos y los métodos invocados :

Evento	**Mientras sucede el evento**	**Una vez sucedido el evento**
Creación		onCreation()
Clonación	onCloning()	onClone()
Envío	onDispatching()	onArrival()
Retracción	onReverting()	onArrival()
Liberación	onDisposing()	
Desactivación	onDeactivating()	onClone()
Activación		onActivation()
Mensaje	handleMessage()	

La clase AgletContext

Un aglet pasa la mayoría del tiempo en un Aglet Contex (un contexto). Se crea en un contexto, queda desactivado en un contexto, y muere en un contexto.

La interfaz AgletContext es usada por un aglet para recibir información acerca de su ambiente y para enviar mensajes al ambiente, incluyendo éste otros agentes activos en ese momento en ese ambiente. El contexto constituye también la manera en que el sistema host se asegura contra aglets maliciosos.

El aglet tiene un método para acceder a su contexto actual :

```
context = getAgletContext();
```

Con tal acceso, puede crear nuevos aglets :

```
context.createAglet(...);
```

Y puede retraer a aglets remotos al contexto actual :

```
context.retractAglet(remoteAgletURL);
```

El aglet puede también obtener una lista de los proxies de los demás agentes presentes en el mismo contexto :

```
proxies = getAgletProxies();
```

El contexto es creado típicamente por un sistema, que tiene un daemon que está al tanto de la red esperando que lleguen aglets, y que inserta a los que lleguen al contexto.

La clase AgletProxy.

El AgletProxy sirve como una cubierta que protege al agente de acceso directo a sus métodos públicos. Un proxy es lo que se obtiene del método createAglet en el contexto :

```
AgletProxy proxy = context.createAglet(...);
```

Algunos de los métodos del proxy, como clone(), dispatch(), dispose(), y deactivate(), se usan para controlar al aglet.

Dos métodos, sendMessage() y sendAsyncMessage(), se usan para enviar mensajes síncronos y asíncronos, respectivamente, al aglet, vía su proxy.

En el paso síncrono de mensajes, el hilo (o thread) que envía el mensaje es detenido hasta que se obtenga un resultado :

```
Object result = proxy.sendMessage(msg);
```

En el paso asíncrono de mensajes, el hilo de envío de mensaje termina inmediatamente y el resultado es subsecuentemente obtenido mediante el objeto *reply* (respuesta) :

```
FutureReply reply = proxy.sendAsyncMessage(msg);
Object result = reply.getReply();
```

La clase Message

Los mensajes se distinguen por un campo String llamado "kind". El segundo parámeto del constructor de un mensaje es un argumento opcional del mensaje :

```
Message miNombre = new Message("mi nombre", "Hermenegildo");
Message tuNombre = new Message("¿tu nombre?");
```

La clase AgletIdentifier

A todo aglet le es asignado un identificador globalmente único, el cual conserva en todo su ciclo de vida. La clase AgletIdentifier es una abstracción conveniente de su identidad.

3.1.4. Eventos en un Aglet

Los eventos principales en la vida de un aglet son los siguientes [3]:

Creación: crea y clona
Liberación: libera
Mobilidad: envía (dispatch) y retrae (retract)
Almacenamiento: Desactivación (deactivate) y Activación (activate)

Creación de un Aglet

Hay dos maneras de crear un aglet : instanciarlo de una clase Aglet, o clonar un aglet existente.

Instanciación de clase: El aglet es creado en un contexto. De ese contexto, el aglet puede obtener un conjunto fijo de servicios, sin importar en qué máquina o sistema operativo se encuentre. Uno de esos servicios es la instanciación de nuevos aglets desde una determinada clase Aglet.

public final AgletContext getAgletContext()
Obtiene el contexto en el que el aglet se está ejecutando.
protected Aglet()
Construye un aglet no inicializado.

public void onCreation(Object init)

Inicializa el nuevo agente.

public void run()

Es el punto de entrada de la ejecución del nuevo aglet. Este método se invoca a partir del éxito del método de creación, envio, retracción o activación del aglet.

Estos métodos son invocados automáticamente por el contexto durante la creación de un aglet. Dos de ellos permiten la customización de la creación del aglet, mediante su sobreescritura : onCreation y run.

Clonación de un Aglet

La clonación es una manera alternativa de crear nuevos aglets. El método clone() de la clase Aglet permite duplicar de manera exacta (excepto por el AgletID) un aglet existente.

public final Object clone()

Clona el aglet y el proxy que contiene el aglet, y retorna el proxy del nuevo aglet.

Dos métodos de la clase Aglet permiten customizar y controlar el proceso de clonación. El primero, onCloning, es llamado por el aglet al intentar la clonación. Es necesario sobreescribir éste método con código adecuado. El segundo, onClone, puede ser usado para inizalizar al clon.

public void onCloning()
public void onClone()

Liberación de un Aglet

Un aglet debe ser liberado cuando haya realizado su tarea y no sea más de utilidad, con el fin de liberar recursos de la máquina en la que corre. El sistema reclamará todas las tareas pertenecientes a un aglet que es liberado. Además, el contexto liberará todos sus recursos que estén ligados al aglet. Al hacer esto, eliminará todas las referencias entre el contexto y el aglet. Sin embargo, no hay garantía de que la memoria utilizada por el aglet sea liberada inmediatamente. El recolector de basura de Java eliminará el aglet cuando todas las referencias a él hayan sido eliminadas, incluyendo las referencias de otros aglets.

Para que un aglet se libere a si mismo, debe llamar al método dispose(), y cualquier operación que el agente deba realizar al ser liberado, debe implementarse en el método : onDisposing().

Movilidad del agente

El envío de un agente a una localidad remota se efectúa mediante el método dispatch(), en el cual se especifica la dirección de salto en la forma de un URL. El URL destino (Universal Resource Locator) debe

contener el nombre del protocolo (en este caso "atp") y la dirección IP o nombre asignado del host remoto al que el agente viajará. El método retract() funciona de manera similar, pero su efecto es contrario, es decir, hace que el agente viaje al sitio donde el método es ejecutado.

Cualquier acción que el agente deba emprender en el momento de ser despachado o retraido, debe implementarse en los métodos onDispatching() y onReverting() respectivamente. El método onArrival() es ejecutado después de que alguna de estas operaciones tiene éxito.

La movilidad en Aglets Workbench es posible gracias a dos facilidades : el Agent Transfer Protocol (ATP) y el Java Agent Transfer and Communication Interface (J-ATCI).

El ATP es un protocolo de nivel de aplicación para agentes distribuidos basado en sistemas de información, y facilita la migración de los agentes de un contexto a otro sobre la red. Basado en el sistema de nombres de Internet, el ATP utiliza el Localizador Universal de Recursos (URL) parta especificar locaciones y constituye un protocolo independiente de plataromas para permitir la transferencia de agentes móviles entre computadoras en red. Aunque este protocolo ha sido distribuido con el Aglets Workbench, su dominio de aplicación no es de ninguna manera exclusivo de los aglets, dado que ofrece la oportunidad de manejar agentes móviles en cualquier lenguaje de programación y una variedad de sistemas de agentes.

Reforzando el ATP a un nivel de comunicación mas alto está J-ATCI, un protocolo de agentes independiente que permite a los aglets moverse y comunicarse a través de la red. J-ATCI es una interface de programación flexible y simple que permite a los programadores desarrollar agentes independientes de plataforma sin necesidad de interconstruir dentro de ellos los protocolos necesarios para establecer la comunicación.

3.1.5. Utilización de Aglets Workbench.

Aglets Workbench es un conjunto de clases e interfaces construidas a partir del sistema de clases que provee Java, por lo que es necesario contar con el JDK (Java Development Kit) o por lomentos el intérprete Java, para poder utilizarlo. La versión Aglets Workbench Alpha5a requiere que se encuentre instalado el JDK 1.1 o una versión posterior.

Aglets Workbench con JDK 1.1 puede correr en SPARC/Solaris2.5, x86/Solaris, Windows95/NT y en AIX4.1.4.

Es necesario especificar ciertas variables de entorno :

(a) AGLET_HOME es el directorio donde se encuentra instalado Aglets Workbench.
(b) PATH debe incluir el subdirectorio AWB\Aglets\bin
(c) CLASSPATH debe incluir el directorio 'lib' donde se encuentran las librerías de los Aglets.

(d) AGLET_PATH es la ruta por defecto para crear un aglet sin codebase.

(e) AGLET_EXPORT_PATH es para archivos que pueden ser traídos de un sitio remoto. En la mayoría de los casos puede ser igual que AGLET _PATH.

En el sistema utilizado en este trabajo se utilizó la siguiente configuración de las variables de entorno :

```
CLASSPATH=c:\Aglets\lib;c:\aglets\public;c:\Aglets\lib\ibm\atp\daemon;
AGLET_HOME=c:\aglets;
PATH=c:\jdk1.1.1\bin;c:\aglets\bin;c:\aglets\lib;c:\Aglets\lib\aglet;c:\Aglet
s\lib\ibm\aglets;
AGLET_PATH=C:\aglets\public;
AGLET_EXPORT_PATH=%AGLET_PATH%
```

Aglets Workbench provee una aplicación llamada Tahiti, que

Para correr Tahiti, el servidor de aglets se ejecuta una archivo de procesamiento por lotes que viene incluido en el paquete :

```
agletsd
ó
agletsd -port 9000 (en plataforma Unix)
```

Aglets Workbench cuenta con una aplicación llamada Tahiti, que funciona como un servidor de agentes. Es posible correr múltiples servidores Tahiti en una sola computadora, asignándoles diferentes números de puerto. Tahiti brinda al usuario una interface para monitorear, crear, enviar y liberar a los agentes, así como para establecer los privilegios de acceso para cada uno de ellos.

Es necesario hacer notar que todo aglet se ejecuta sobre un *contexto* (AgletContext) que funciona como contenedor de agentes, como interfaz entre el agente y la máquina, y como medio de comunicación entre agentes, y no es posible ejecutar un aglet sin existir por lo menos un contexto. En ese sentido, la utilidad primordial de Tahiti radica en que brinda con contexto de ejecución sobre el cual pueden existir los agentes.

Para correr Tahiti, es necesario ejecutar un archivo de procesamiento por lotes (agletsd.bat), que colocará el servidor Tahiti en el puerto 434 (reservado para los agentes móviles), y hará que su interfaz aparezca en pantalla.

Para correr un segundo servidor Tahiti en la misma computadora, se ejecuta el archivo agletsd especificando el puerto en el que quedará colocado el nuevo daemon, por ejemplo : agletsd -port 500. Es posible escoger cualquier puerto que no esté siendo ocupado en ese momento por alguna otra aplicación o el sistema operativo.

3.2. Estructura Funcional del Sistema Multiagentes

El propósito de ésta sección es mostrar la estructura funcional del sistema multiagentes, es decir, la forma en que los agentes se encuentran relacionados unos con otros, y la manera en que este conjunto de relaciones agente-agente les permite proporcionar una solución global al problema.

3.2.1. Alcances y limitaciones

Existe una gran variedad de consideraciones que se deben hacer cuando se decide dotar a un sistema multiagentes de inteligencia. Si bien existe una vasta cantidad de documentación disponible para cada una de ellas, se presentan diversos problemas con su implementación práctica :

- En general, existen implementaciones prácticas para un subconujunto reducido de los conceptos teóricos que se abordan en la inteligencia artificial distribuida.
- Tales implementaciones tienen fines experimentales. Su meta es probar teorías acerca del comportamiento grupal inteligente, que en general tienen su origen en la intuición y que resultan en mayor o menor medida ciertas.
- Los recursos de cómputo disponibles en el presente resultan insuficientes si se desea dotar a los programas de un comportamiento inteligente complejo. En el mejor de los casos se

obtienen aplicaciones que consumen demasiada memoria y tiempo de procesamiento. La significación de este problema se acentúa si consideramos que los agentes ocupan recursos de cómputo ajenos al propietario, y que puede existir un grupo de agentes por cada usuario de internet.

Podemos identificar aspectos de la inteligencia del sistema que conforman sus deficiencias actuales, pero que constituyen sin embargo, sus potenciales mejoras.

Primeramente, es evidente que un factor escencial en la facilidad de uso de un sistema de información, es su conveniente abstracción. En ese sentido podemos decir que el sistema propuesto permite alcanzar un nivel de abstracción superior que otros sistemas de búsqueda de información, en virtud de que los resultados que éste brinda están en función de las preferencias del usuario (que pueden representarse con mayor o menor complejidad), en lugar de simples búsquedas en listas de títulos.

Sin embargo es fácil ver que es posible alcanzar varios niveles de abstracción adicional :

- Un primer paso en ese sentido sería mostrar al usuario el documento propuesto en lugar de solo su URL (se sugiere leer la documentación de Aglets Workbench sobre como correr un servidor de agentes en conjunción con el web browser).

- En un segundo nivel, el sistema multiagentes no muestra los documentos al usuario, sino que los lee, los interpreta y presenta al usuario aquella información que considere relevante.
- Finalmente, el sistema podría no solo mostrar la información obtenida, sino utilizarla para resolver problemas de alto nivel planteados por el usuario al sistema [19].

Otro aspecto importante del sistema que requiere un estudio mas completo es el proceso de comunicación e interacción entre agentes. Actualmente, poco se ha alcanzado en la estandarizarización del proceso de comunicación entre los agentes, sin embargo es evidente que a medida que esta tecnología se difunda y alcance niveles comerciales de utilización, un mecanismo uniforme de comunicación que permita a sistemas heterogéneos interactuar, será de vital importancia (el lenguaje de comunicación interagentes que ha logrado mayor aceptación a la fecha es KQML [29], Knowledge Query and Manipulation Language).

Dado que el sistema multiagentes aquí propuesto orienta todas las comunicaciones hacia el interior del mismo sistema, la utilización de un lenguaje y protocolo estándar no fue considerado un punto crucial, como lo sería si se quisiera que el sistema interactuara con otros agentes o sistemas multiagentes distintos.

Por último, es probable que agentes como el agente Descubridor de este sistema, que tienen que enfrentarse a situaciones muy variadas,

requieran de un comportamiento con un grado mayor de complejidad, sin comprometer por ello su agilidad y manteniendo su bajo consumo de recursos de cómputo.

En ese sentido, dos aspectos perfectibles del agente sobresalen. Primero, es posible que el agente tenga que competir con otros agentes, pertenecientes a otros usuarios, por recursos e información, por lo que los mecanismos de negociación serán en el futuro una pieza clave en la implementación de sistemas comerciales. Segundo, es necesario proveer al agente de un mecanismo de aprendizaje que evite que el diseñador del agente esté obligado a incrustar todo el conocimiento (específico de sus funciones) que el agente requiere.

En el caso específico del agente Descubridor, el mecanismo de aprendizaje debe ser tal que construya, modifique y extienda la biblioteca de planes mediante la síntesis de nuevos planes [21][22]. Estos nuevos planes pueden generarse mediante diversas metodologías, de las que podemos mencionar las siguientes :

- Construcción aleatoria de planes, ejecución de los mismos, y evaluación de los resultados.
- Simulación de planes basada en el conocimiento de los efectos de las acciones en el ambiente. Esta simulación también puede incluir las reacciones de otros agentes (a partir de un modelo de tales agentes) y la evolución natural del ambiente.
- Comunicación de planes entre agentes.

- Evolución (programación genética).

3.2.2. Utilización del sistema

Desde el punto de vista del usuario, el sistema no es mas que un agente que trabaja para él, (en realidad se trata del agente Central, como veremos mas adelante), que eventualmente le muestra documentos nuevos que posiblemente le interesen, y le pide que los evalúe de acuerdo al grado de interés que estos poseen. Por otra parte, el usuario puede hacer preguntas explicitas al agente, ante lo cual, el agente responderá de la siguiente manera :

- Mostrará al usuario documentos relacionados con la pregunta, pero que ya había mostrado anteriormente, y además preguntará si esos documentos son sufiencientes
- Si los documentos mostrados no son suficientes, el agente comenzará en las próximas horas a traer y a sugerir documentos adicionales para intentar satisfacer la pregunta.

3.2.3. Estructura y Organización

En un ambiente multiagentes, la estructura es el patrón de relaciones de información y de control que existen entre los agentes, y la distribución de la capacidad de resolver problemas entre ellos [22].

El sistema multiagentes que nos ocupa costituye una organización, dado que se encuentran conectados de alguna manera (arreglados sistemáticamente) y sus actividades combinadas resultan en algo mejor (mas armonioso), que si no estuvieran conectados. Como tal, esta organización consiste de :

- Un grupo de agentes
- Un conjunto de actividades desarrolladas por los agentes
- Un conjunto de conexiones entre los agentes
- Un criterio de evaluación para las actividades combinadas de los agentes.

La estructura del sistema debe asignar *roles* y *relaciones* de manera que se puedan satisfacer las siguientes condiciones :

Cobertura : cada porción necesaria de la información requerida por el usuario debe ser parte del dominio de al menos un agente.
Conectividad : los agentes deben interactuar de tal manera que las actividades cubiertas sean desarrolladas e integradas en una solución global.
Capacidad : La cobertura y la conectividad deben ser alcanzables dentro de las limitaciones en los recursos de comunicación y computación.

Los roles y relaciones en nuestro sistema multiagentes han sido fijados de antemano, es decir, no forman parte de las capacidades de

adaptación y aprendizaje del sistema, por lo que la capacidad de solución de problemas distintos al que nos ocupa, puede ser bastante limitada, sin embargo, el diseño de la estructura del sistema satisface las tres condiciones arriba mencionadas.

Con el objetivo de proveer a la red y a sus depósitos de información, de niveles graduales de abstracción, la estructura del sistema se ha diseñado de forma jerárquica, en la que cada nivel se ocupa de manejar la información en etapas diferentes de abstracción, y los roles y relaciones entre los agentes quedan definidos por esa estructura (figura 3.1).

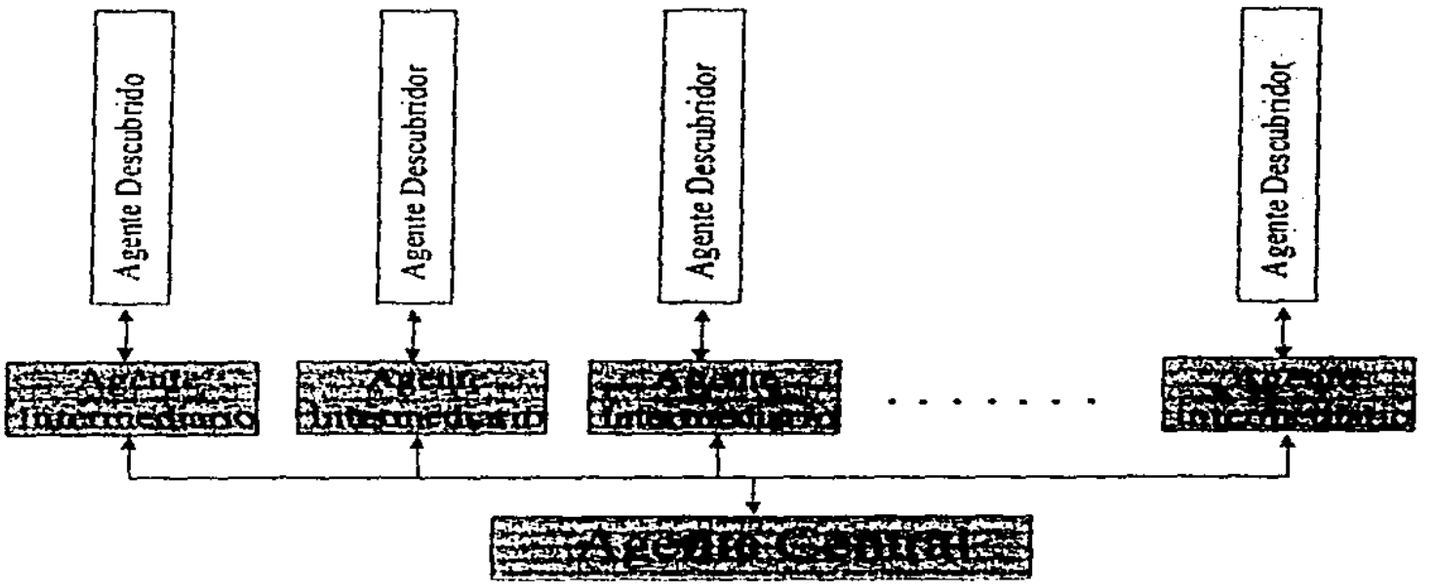


Diagrama Jerárquico del Sistema Multiagentes

figura 3.1

Analizaremos esta estructura comenzando por los agentes Intermediarios.

Dado que las preferencias del usuario están constituidas, casi siempre, no por un solo tema preferido, sino por un conjunto de temas diversos, sin ninguna relación necesaria entre ellos, es conveniente hacer que el sistema disponga de diferentes partes independientes que se especialicen en cierta área de los intereses del usuario. De esta manera la búsqueda de documentos lleva a soluciones mas precisas en menor tiempo. Por tal motivo, el sistema multiagentes asigna un agente a cada tema de interés, y puede haber tantos o tan pocos agentes como sea necesario. Éstos son los agentes Intermediarios.

Cada agente Intermediario tiene como función ajustarse a una de las áreas de interés, y mantener esa adaptación a través del tiempo aunque los intereses del usuario cambien. Estas funciones suponen las siguientes características :

- Debe responder de manera inmediata preguntas explícitas por parte del usuario.
- Capacidad para recibir retroalimentación inmediata por parte del usuario.
- Capacidad para modificar su concepción del usuario.
- Debe poder almacenar su conocimiento del usuario en memoria secundaria, de manera que éste sobreviva a los períodos de inactividad de la computadora donde se aloja el sistema.

Estas características del agente Intermediario muestran que sería poco práctico hacer que los mismos agentes Intermediarios sean los encargados de viajar a cada uno de los repositorios de información. Esa es la función del agente Descubridor.

El agente Descubridor es considerablemente mas pequeño y almacena cantidades pequeñas de información, características ideales para un programa que tiene que ser transmitido entre diferentes sitios y que ocupará recursos de cómputo en servidores ajenos. Un agente Descubridor se encuentra siempre al servicio de sólo un agente Intermediario, y ambos contienen el mismo conocimiento, solo que a diferencia de éste último, el agente Descubridor :

- No recibe consultas directas por parte del usuario.
- No se encarga de recibir la evaluación de documentos por parte del usuario.
- No manipula el conocimiento que se tiene del usuario.
- No accesa nunca ningún medio de almacenamiento secundario.

Los agentes Intermediarios no obtienen información de la red, sino de los documentos que les son traídos por los agentes Descubridores que se encuentran a su cargo.

Por último, con el fin de presentar al usuario una interfaz única, que simplifique la utilización del sistema, existe un agente Central, que es el

encargado de recopilar los resultados parciales obtenidos por los agentes Intermediarios, y presentarlos en conjunto al usuario, pedirle la evaluación de los documentos y permitir la consulta directa, y redistribuir los documentos evaluados entre los agentes Intermediarios. El agente Central se encarga también de repartir entre los agentes Intermediarios el trabajo que se tiene que hacer para satisfacer una consulta.

Es preciso señalar que debido a la naturaleza distribuida de la estructura del sistema y por el bajo nivel de acoplamiento entre agentes en el mismo nivel de la jerarquía, es muy fácil escalar el sistema tanto en profundidad como en anchura.

Un aumento en la anchura de la jerarquía se da cuando la diversidad de los temas preferidos por el usuario así lo amerita. Por otra parte, un aumento en la profundidad de la estructura requiere de cierto esfuerzo por parte del diseñador, pero es aún relativamente fácil, y permitiría una abstracción mas conveniente de las tareas a realizar por los agentes en cada nivel jerárquico. Pero la principal conveniencia de un aumento en los niveles de abstracción del sistema, es que permitiría la inclusión de tantas tareas, específicas y generales, como se requiera, y que quedan fuera del alcance de éste trabajo.

3.2.4. Coordinación, cooperación y comportamiento coherente

La coordinación del comportamiento del grupo de agentes es un componente fundamental en el buen desempeño del sistema. La coordinación entre agentes es especialmente útil cuando en el sistema existe una gran necesidad de *asignar recursos escasos* y *comunicar resultados intermedios* [22].

Para explicar cómo funciona el mecanismo de coordinación en nuestro sistema multiagentes, es necesario entender que la estructura jerárquica del mismo plantea dos formas posibles de interacción entre agentes : a) en anchura y b) en profundidad.

a) Los agentes que pertenecen a un mismo nivel jerárquico del sistema presentan un bajo nivel de acoplamiento, y ello se debe a las siguientes razones :

- La realización exitosa de las tareas que un agente pretende llevar a cabo, no depende de las tareas que realice o no otro agente en el mismo nivel.

- Casi no existen recursos limitados por los cuales los agentes deban competir, y por los cuales deban intercambiar información y llegar a un arreglo.

De hecho, solo existe un recurso por el cual competir : un tema sobre el cual trabajar. Esto se deriva del hecho de que dos o mas agentes trabajando sobre un mismo tema, significan no solo esfuerzo redundante, sino la inconvenciencia de mostrar los mismos documentos al usuario varias veces. Es al evitar esa redundancia donde la coordinación entre los agentes es crucial.

En el proceso de coordinación entre los agentes, el usuario juega un papel primordial. El usuario, al evaluar cada uno de los documentos que el sistema le propone, hace posible un esquema de coordinación por supervisión directa, mediante el cual puede hacer que un agente que está trabajando sobre un tema que otro agente también está cubriendo, modifique su conocimiento sobre el usuario y trate de buscar un nuevo nicho de trabajo.

Este proceso requiere que el usuario evalúe de manera negativa cualquier documento que el sistema le presente cuando el sistema ya se lo haya mostrado antes.

La cooperación entre agentes, por los motivos arriba mencionados, es igualmente sencilla. Se asume una postura benevolente para cada agente, lo que implica que un agente entrega información veraz a otro en el mismo nivel si considera que puede serle útil. De esta manera, cuando un agente Intermediario adquiere nuevos documentos en un momento en el que las preferencias del usuario respecto a ese tópico ya han cambiado, el agente distribuye esos documentos entre el resto

de los agentes intermediarios, ya que posiblemente le interesen a algunos de ellos.

b) La coordinación entre agentes relacionados de manera vertical en la jerarquía, es decir, la coordinación entre el agente Central y los Intermediarios, y la coordinación entre éstos últimos y los Descubridores, tiene una importancia mayor.

Los agentes Intermediarios y el agente Central realizan las tareas que les corresponden a partir de eventos que provienen del ambiente exterior, y que pueden ser originados por el sistema, el usuario, u otros agentes, es decir, son agentes controlados por eventos.

Un agente puede darse cuenta de los eventos originados en el exterior por otros agentes, únicamente mediante el paso de mensajes. Esto es, un mensaje enviado de un agente a otro constituye un evento para el agente receptor. Esto presupone que todo agente conoce de antemano cuáles son los mensajes que otro agente puede interpretar, y cómo será interpretado. Estos mensajes juegan un papel central en el mecanismo de coordinación del sistema, porque por otra parte, no solo hacen posible que un agente se entere de eventos suscitados en otro agente, sino que permiten transmitir información entre ellos.

La manera en que el flujo de información tiene lugar en el sistema, se ilustra a continuación. En la figura 3.2 se ilustra el flujo local de información, y en la figura 3.3 el flujo de información en un host remoto.

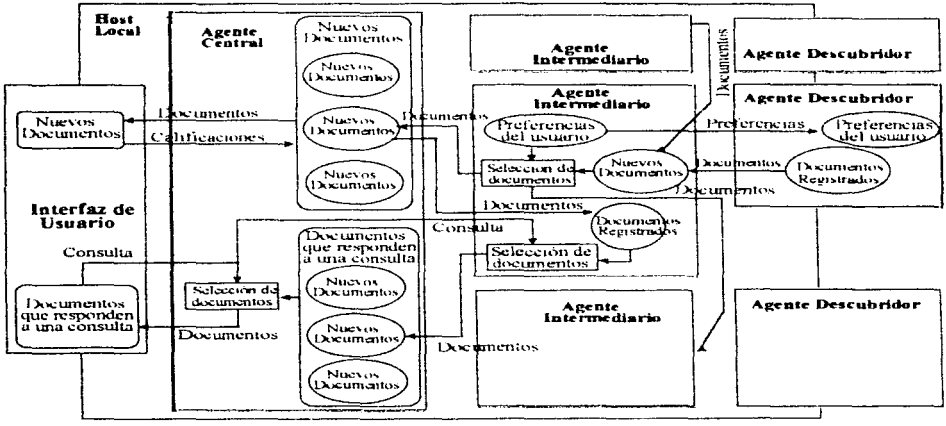


figura 3.2

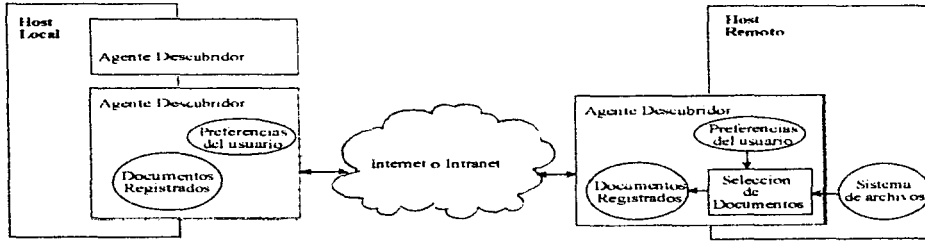


figura 3.3.

3.2.5. Planeación

El proceso de planeación en el sistema multiagentes se realiza de manera distribuida. Es el agente Central el que, a partir de los requerimientos del usuario, determina el crecimiento o contracción del sistema multiagentes a corto plazo, es decir, la variedad de temas sobre los que es necesario trabajar. Y de igual forma, es responsabilidad de los agentes Intermediarios planear el crecimiento o contracción de su plantilla de agentes correspondiente.

Sin embargo, debido a la autonomía con la que necesariamente deben contar los agentes, cada uno de ellos desarrolla sus propios planes, y de ninguna manera un agente puede influenciar los planes de otro. De igual manera, no existe una planeación global a la cual cada agente deba someterse ni con la cual deba colaborar.

La planeación que lleva a cabo un agente Descubridor durante su ciclo de vida, merece un tratamiento especial, por lo que se hablará de ella en la sección sobre la arquitectura de los agentes.

3.2.6. Comunicación entre agentes

La comunicación entre agentes les permite intercambiar información y coordinar sus actividades [22]. En nuestro sistema, los agentes se comunican mediante el intercambio directo de mensajes [3]. Aunque se utiliza únicamente la comunicación síncrona, es decir, aquella en que el

agente receptor recibe el mensaje al mismo tiempo en que el remitente lo envía, es posible utilizar también la comunicación asíncrona.

Aglets Workbench provee una plataforma en la cual los agentes no necesitan conocerse unos a otros al momento de compilación para comunicarse. Este sistema permite que agentes que fueron creados en organizaciones diferentes puedan intercambiar mensajes, y constituye un modelo de comunicación mas rico, flexible y extensible que el usado con la invocación de métodos tradicional.

Los aglets soportan un marco de intercambio de mensajes orientado a objetos que es :

- Independiente de locación
- Extensible
- Síncrono / Asíncrono

Los mensajes intercambiados tienen una estructura fija y muy simple. Cada mensaje consta solamente de tres partes : el tipo de mensaje, un argumento, y una marca de tiempo en el que el mensaje fue enviado. El tipo de mensaje permite al agente receptor, identificar la clase de información que el mensaje transporta, y el argumento es el portador de la información en si. El argumento del mensaje puede ser usado de tres maneras distintas (figura 3.4):

a) Puede enviarse un mensaje sin ningún tipo de argumento

b) El argumento puede ser un objeto

c) El argumento puede ser una tabla de objetos

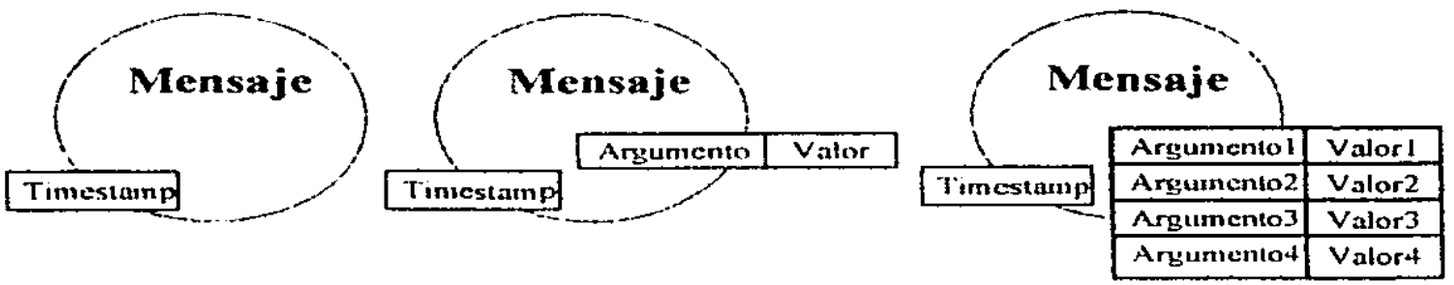
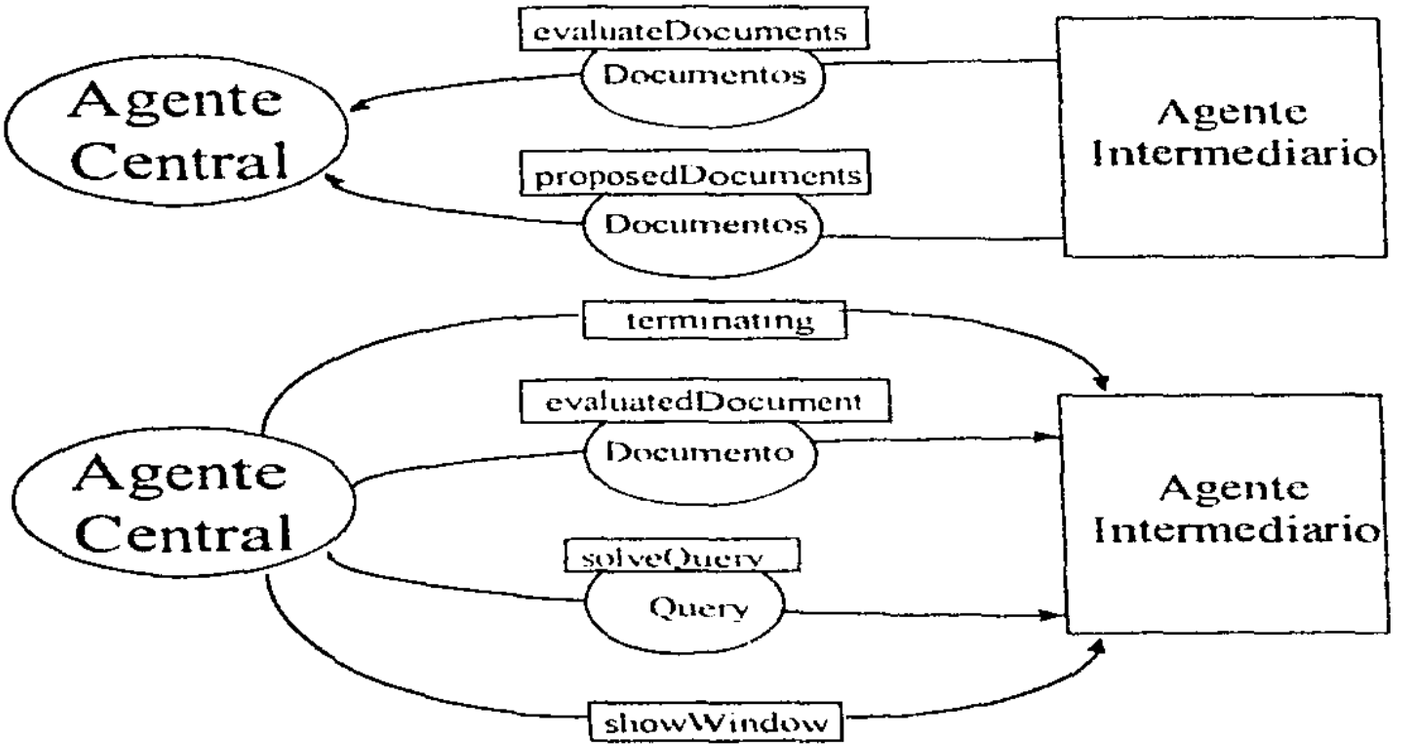


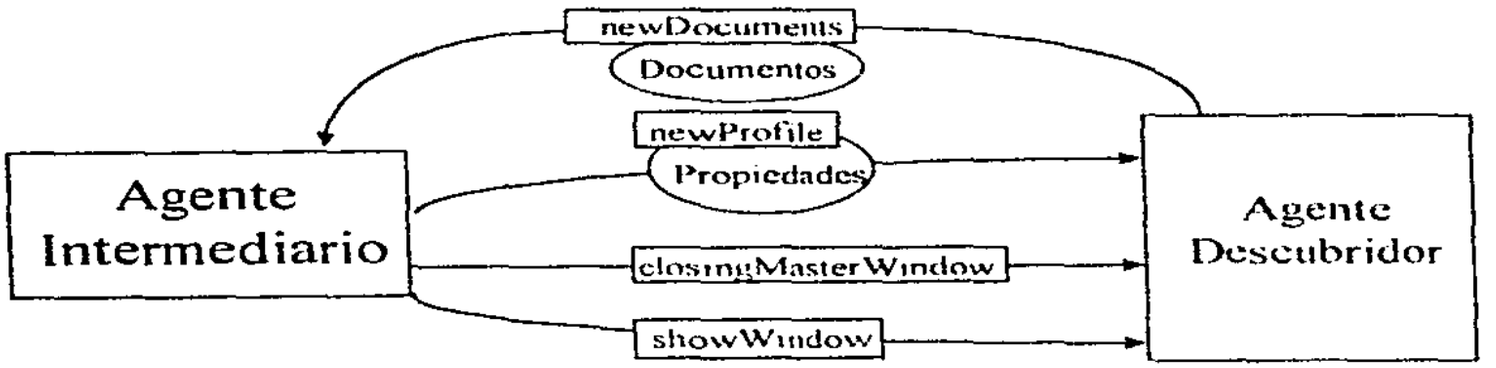
figura 3.4.

El intercambio de mensajes entre los agentes se realiza casi exclusivamente de manera vertical en la jerarquía del sistema multiagentes, es decir, la mayoría de los mensajes pasan del agente Central al Intermediario y del agente Intermediario al Descubridor, y en sentido contrario. Esta estructura de comunicación permite mantener la independencia de cada uno de los grupos de agentes que conforman el sistema.



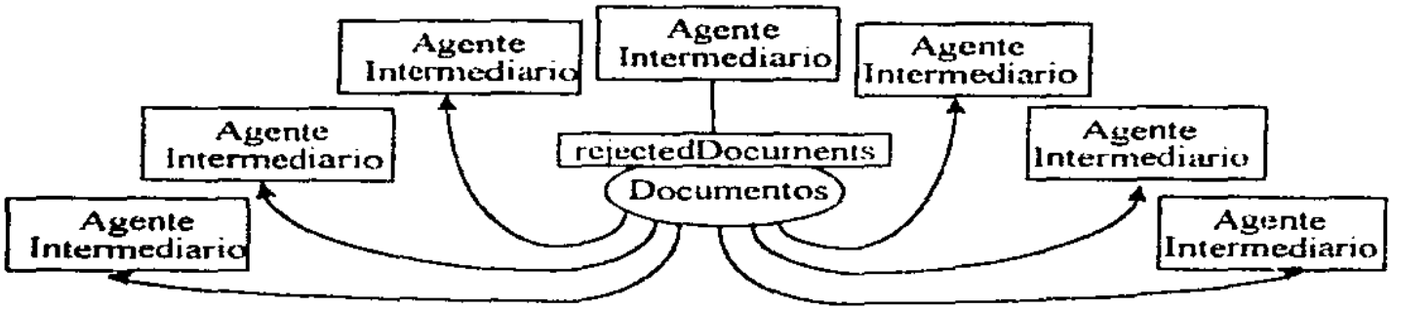
Intercambio de mensajes entre el agente Central y el Intermediario

figura 3.5.



Intercambio de mensajes entre el agente Intermediario y el Descubridor

figura 3.6.



Intercambio de mensajes entre agentes Intermediarios

figura 3.7.

Es importante hacer notar que el paso de mensajes entre los agentes no permite la invocación directa de métodos del agente receptor, ni acceso a sus variables internas. Esto se traduce en el aseguramiento de la autonomía de cada agente, ya que cada uno de ellos determina por su propia cuenta, no solo qué mensajes enviar, a quién y cuándo, sino también qué acciones emprender a partir de la recepción de un mensaje.

El paso de un mensaje a un agente se realiza a través de su apoderado (proxy), por lo que es necesario conocer la identidad del agente receptor (figura 3.8).

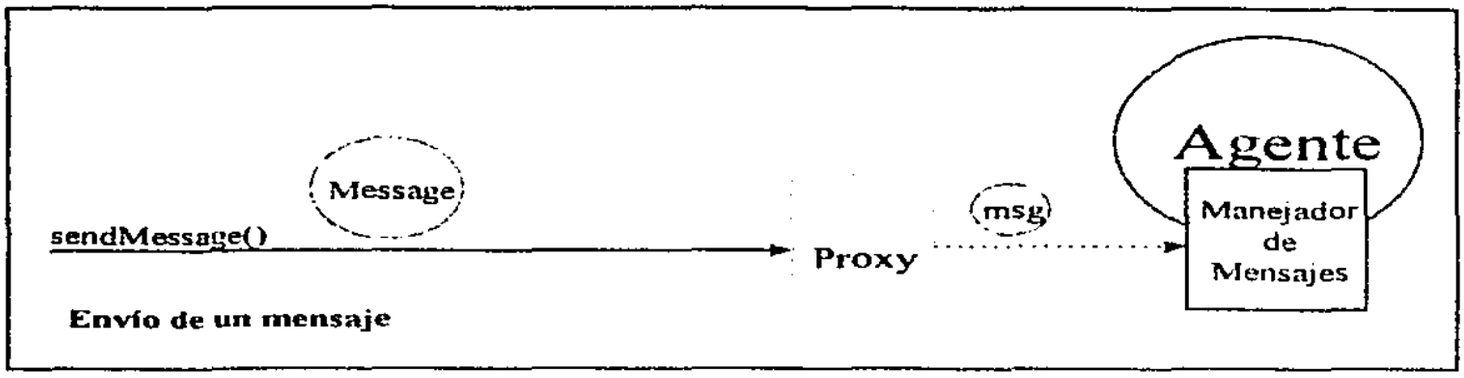


figura 3.8.

El agente destinatario cuenta con un mecanismo receptor de mensajes, que no es otra cosa que un método llamado handleMessage() que es invocado cuando el agletProxy recibe un nuevo mensaje, y que determina las acciones a tomar para cada tipo posible de mensaje.

```
public boolean handleMessage(Message msg) {
    if (msg.sameKind("terminating")) {
        dispose();
        return true;    // Si, mensaje recibido y utilizado
    } else
        return false;   // No, no utilicé el mensaje
    }
}
```

Este método permite también enviar inmediatamente una respuesta al agente emisor del mensaje (figura 3.8):

```
public boolean handleMessage(Message msg) {
```

```
    if (msg.sameKind("Cuál es tu nombre ?")) {
       msg.sendReply(miNombre);
       return true;
    }
    return false;
}
```

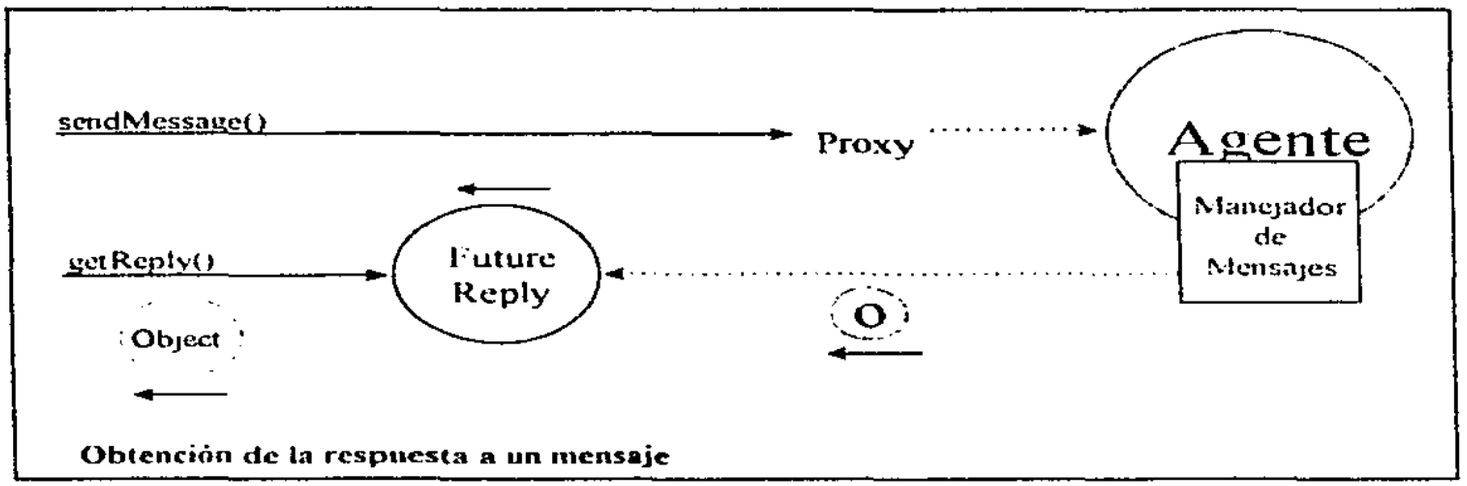


figura 3.8.

Como se mencionó antes, el sistema que nos ocupa utiliza solamente la comunicación síncrona, y hasta ahora hemos visto la manera en que ésta se realiza cuando un agente trata de comunicarse con otro. Sin embargo, existe otra forma de comunicación por paso de mensajes de manera síncrona, que es la difusión del mensaje (o multicasting), que permite que el mensaje sea recibido por todos los agentes que se encuentran en el mismo contexto que el agente emisor.

Cuando un agente difunde un mensaje, no necesita conocer, al contrario de la comunicación agente-agente, la identidad de sus destinatarios. Esta característica es especialmente útil en las siguientes circunstancias :

- Cuando un agente Descubridor regresa a la computadora hogar y debe anunciar su llegada, pero durante su ausencia, el agente Intermediario al cual sirve fue eliminado. La difusión del mensaje permite que éste, y la información que contiene, alcance al agente Intermediario que ahora ocupa ese puesto.
- Cuando el agente Central desea hacer pública una consulta del usuario a todos los agentes Intermediarios.
- Cuando un agente Intermediario desea transmitir cierta información al agente Central, no es necesario conocer la identidad de éste último, ya que solo existe uno, y por lo tanto al difundir el mensaje se obtendrá el mismo resultado.

Es necesario aclarar que las afirmaciones anteriores solo son váildas si en el contexto de ejecución actual se encuentra operando un solo sistema multiagentes, ya que de otra manera, el mensaje difundido a partir de un agente Intermediario para el agente Central, por ejemplo, podría alcanzar a los agentes Centrales de los otros sistemas, provocando el funcionamiento incorrecto de éstos. Por este motivo, los mensajes son enviados al agente Central en una modalidad punto a punto.

Un agente que desea recibir mensajes que son difundidos por otros agentes debe :

1. Subscribirse a esos mensajes
2. Implementar un método messageHandler() que reciba los mensajes y los utilice

Usualmente un agente se subscribe a un mensaje en el momento de su creación :

```
public void onCreation(Object o) {
     subscribe("newDocuments");
}
```

Y el manejador de mensajes se encarga de recibir el mensaje y decidir que acciones emprender :

```
public boolean handleMessage(Message msg) {
    if (msg.sameKind("newDocuments")) {
        ...
        return true;
    }
}
```

A continuación se presenta una tabla que contiene los tipos de mensaje utilizados en el sistema, así como una descripción de su significado.

evaluateDocuments	Conjunto de documentos	Punto a punto	Indica al agente Central que ha obtenido nuevos documentos, y que a su vez pida al usuario que los evalúe.
ProposedDocuments	Conjunto de documentos	Punto a punto	Comunica al agente Central el conjunto de documentos que satisfacen una consulta previamente hecha.
Terminating	ninguno	Multicast	Indica a todos los agentes Intermediarios que el sistema está a punto de finalizar su ejecución.
EvaluatedDocument	Documento	Punto a punto	Transfiere un documento calificado por el

			usuario al agente Intermediario que originalmente lo obtuvo.
SolveQuery	Consulta	Multicast	Indica a todos los agentes Intermediarios que el usuario ha hecho una consulta, y les transfiere las características de ésta para que puedan responderla.
ShowWindow	ninguno	Punto a punto	Indica a un agente Intermediario que el usuario desea ver su ventana.
RejectedDocuments	Conjunto de documentos	Multicast	Difunde entre todos los agentes Intermediarios los documentos recién adquiridos que no le son útiles a uno de ellos.

NewDocuments	Conjunto de documentos	Multicast	Transfiere a un agente Intermediario los nuevos documentos que un agente Descubridor a su servicio ha obtenido.
NewProfile	Propiedades	Punto a punto	Notifica al agente Descubridor las preferencias actualizadas del usuario.
ClosingMasterWindow	ninguno	Punto a punto	Notifica a un agente Descubridor que el agente Intermediario al cual sirve, ha cerrado su ventana.
ShowWindow	ninguno	Punto a punto	Notifica a un agente Descubridor que el usuario desea ver

su ventana.

De esta manera tenemos que los agentes Intermediarios deben subscribirse a los siguientes mensajes :

- newDocuments
- solveQuery
- terminating
- rejectedDocuments

3.2.7. Arquitectura de los agentes

En esta sección estudiaremos la manera en que las distintas partes de un agente se encuentran organizadas, es decir, cómo se integra la arquitectura interna de cada agente. Dicha organización de componentes es un factor decisivo en la funcionalidad individual del agente, y por lo tanto determina el comportamiento del sistema multiagentes en su totalidad [22].

Es necesario aclarar que la arquitectura es idéntica en agentes que se encuentran en un mismo nivel de la estructura jerárquica del sistema, en tanto que es radicalmente diferente de la de agentes en otros niveles. Esto se debe, por supuesto, a que los agentes en un mismo nivel jerárquico desarrollan la misma actividad, diferente de las actividades que se llevan a cabo en cualquier otro nivel.

3.2.7.1. Agente Central

El funcionamiento del agente Central se encuentra orientado a eventos, es decir, no actúa por iniciativa propia, sino solamente como respuesta a un estimulo externo. Dichos estímulos externos o eventos pueden ser originados por otro agente, por el usuario, o por el servidor de agentes (Tahiti aglets server) y pueden causar una reacción por parte del agente central.

El agente Central es un agente que actúa por simple reflejo, es decir, que pasa directamente de las percepciones a las acciones, sin pasar por ningún proceso deliverativo (figura 3.9) [21]. El siguiente diagrama esquemático muestra el proceso de decisión de un agente Central :

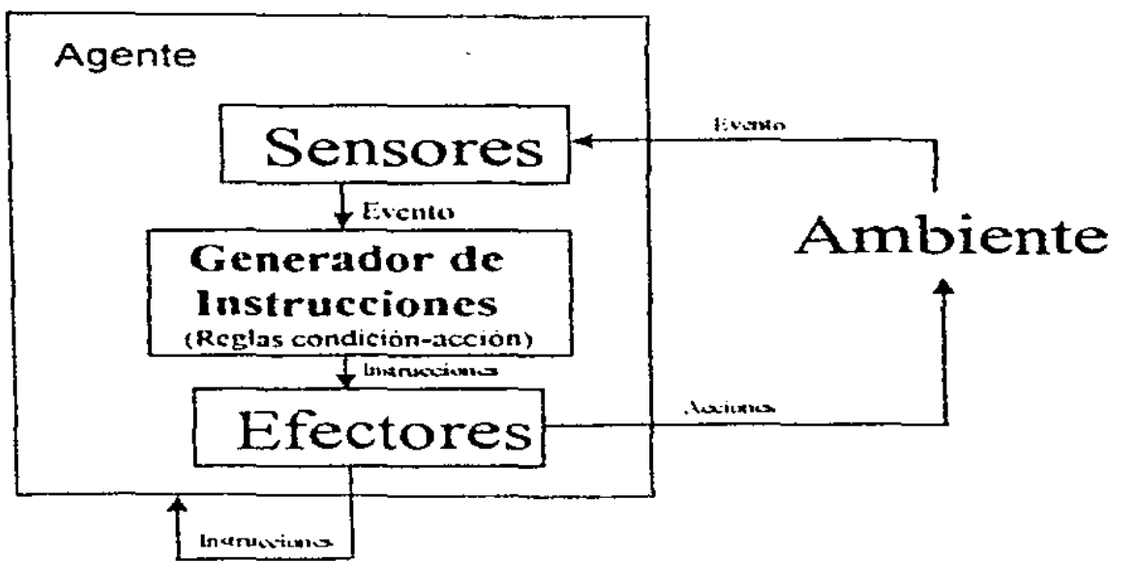


figura 3.9.

Los sensores son los mecanismos que permiten al agente Central ser consciente de los eventos generados en el exterior y que le afectan directamente, y consisten de tres partes :

- El manejador de mensajes
- La ventana de usuario
- Las interfaces de movilidad y persistencia.

El manejador de mensajes indica al agente cuando éste ha recibido un nuevo mensaje proveniente de otro agente, en tanto que la ventana de usuario notifica al agente cuando un evento ha sido originado por éste.

Las interfaces de mobilidad y persistencia indican al agente cuando un evento de creación, desctrucción, movilidad, activación o desactivación, ha sido originado en el servidor de agentes.

Los efectores, por otra parte, están constituidos por métodos que pueden realizar acciones que tienen su efecto ya sea en el ambiente, o en el propio agente y sus componentes internos.

Los estímulos externos provocados por otro agente toman la forma de mensajes dirigidos al agente Central, de los cuales se habló en la sección de comunicación entre agentes, en tanto que los eventos generados por el usuario se originan en la interfaz gráfica del agente.

A continuación se muestra una tabla que relaciona el mensaje recibido con las acciones correspondientes tomadas por el agente Central :

Mensaje	**Acciones**
evaluateDocuments	Incorpora los documentos recibidos a su base interna de documentos. Despliega los nombres de los documentos en la ventana. Notifica al usuario que tiene nuevos documentos que sugerirle.
proposedDocuments	Incorpora los documentos propuestos a su base interna de documentos generados por consultas. Notifica al usuario que ha recibido documentos que responden a una consulta hecha por él.

La siguiente tabla muestra los eventos generados por el usuario y las acciones correspondientes del agente Central :

Evento	**Acciones**
Consulta	Ofrece al usuario una ventana para que formule su pregunta. Notifica a todos los agentes Intermediarios sobre la consulta que el usuario ha hecho.

Ver documentos propuestos	Muestra una ventana con los documentos propuestos que contestan una consulta.
Necesito mas documentos	Crea un nuevo agente Intermediario que trabaje sobre el tema consultado.
Calificar documentos	Permite que el usuario asigne una calificación a uno de los documentos sugeridos. Regresa el documento calificado al agente Intermediario que lo obtuvo.
Ver agente	Notifica al agente en cuestión que el usuario desea ver su ventana.
Ver información sobre el agente	Muestra información técnica sobre el agente en cuestión.
Terminar	Cierra la ventana. Notifica a todos los agentes Intermediarios que el programa está terminando. Actualiza las propiedades del programa. Salva las propiedades en el disco.

La tabla a continuación muestra la reacción del agente Central a los eventos originados por el aglets server, es decir, la plataforma sobre la cual corren los agentes :

Evento	**Acciones**
Create	Crea su ventana Carga las propiedades del sistema en

	memoria. Crea los agentes Intermediarios con los que cuenta el sistema.
Dispatch	Arroja una excepción de seguridad que será manejada por el aglets server (Tahiti)
Revert	Arroja una excepción de seguridad que será manejada por el aglets server (Tahiti)
Deactivate	Oculta la ventana
Activate	Muestra o trae al frente la ventana
Dispose	Cierra la ventana. Notifica a todos los agentes Intermediarios que el programa está terminando. Actualiza las propiedades del programa. Salva las propiedades en el disco.

Estructura funcional del agente Central :

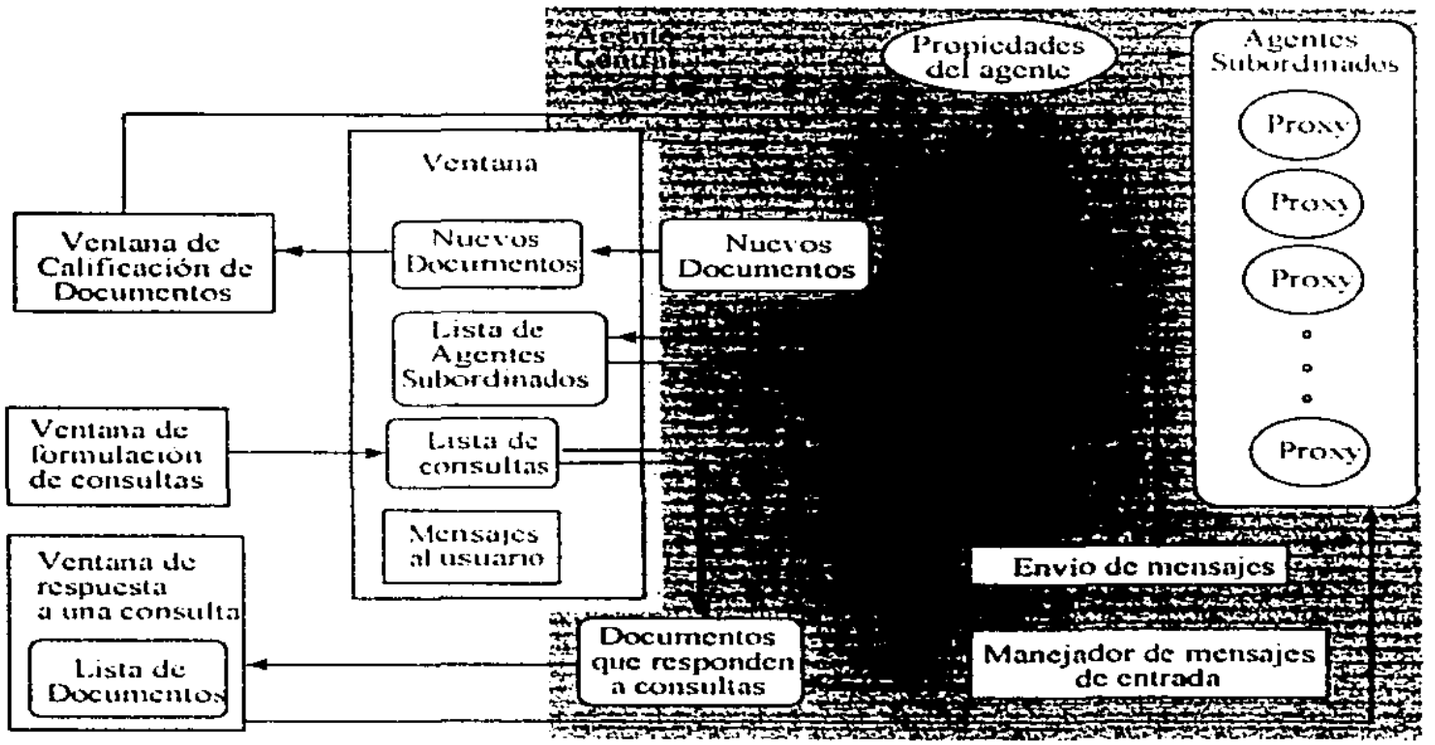


figura 3.10.

Dado que el agente Central es un agente orientado a eventos, el control del funcionamiento del programa queda a cargo de dos estructuras : el manejador de mensajes de entrada, y la interfaz de usuario (figura 3.10).

El manejador de mensajes de entrada se encarga de incorporar a las dos bases internas de documentos, los documentos recibidos de agentes intermediarios a través de mensajes, así como de organizarlos de acuerdo al agente que los obtuvo y a la consulta que los generó, si es el caso.

La interfaz de usuario cuenta con cuatro ventanas cuyos propósitos se describen a continuación:

La ventana principal :

- Muestra los nombres de los agentes subordinados y permite pedir al agente Central que llame a alguno de ellos o información relativa a él.
- Muestra los documentos que el agente Central sugiere en ese momento.
- Muestra las consultas hechas por el usuario y permite pedir al agente Central que muestre los documentos que responden a alguna de ellas.
- Despliega mensajes del agente Central para el usuario.

Interfaz de usuario del agente Central :

La siguiente figura muestra la ventana principal del agente Central :

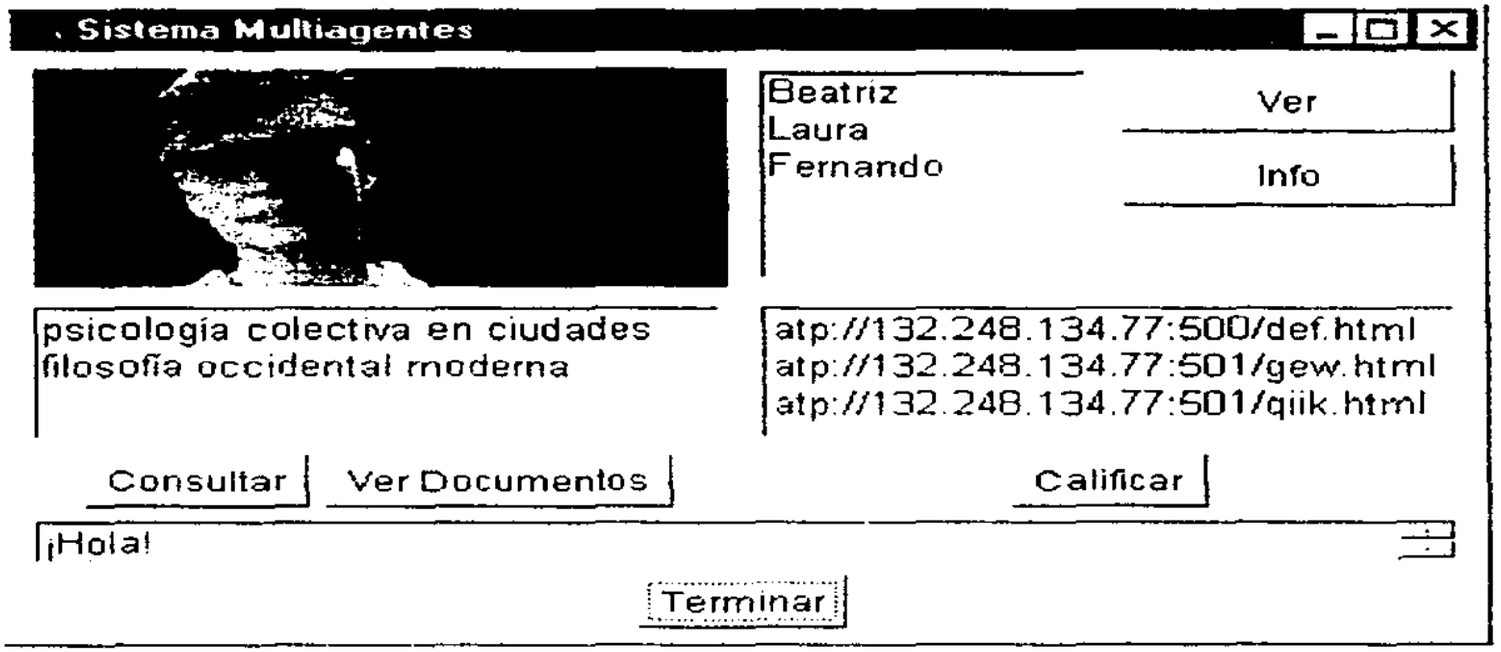


figura 3.11.

La ventana de calificación de documentos :

- Permite calificar uno de los documentos sugeridos por el agente Central.
- Indica al agente Central que el documento ha sido calificado y que puede enviarlo de regreso al agente que lo obtuvo.

La ventana de formulación de consultas

- Permite formular al agente Central una consulta explícita.

La ventana de respuesta a una consulta

- Despliega en pantalla los nombres de los documentos que responden a una consulta explícita por parte del usuario.
- Indica al agente central si el usuario necesitará aún mas documentos relacionados con el tema.

Por otra parte, el agente Central cuenta con cuatro repositorios internos de intormación :

- Nuevos documentos : Es el conjunto de documentos que ha recibido de agentes intermediarios, y que esperan para ser evaluados por el usuario. Estos documentos se encuentran organizados de acuerdo al agente que los obtuvo, de manera que el agente Central realice fácilmente la tarea de redistribución de los mismos a los agentes correspondientes una vez que hayan sido evaluados por el usuario.
- Documentos que responden a consultas : Es el conjunto de documentos que han sido recibidos de otros agentes como respuesta a consultas explícitas del usuario. Se encuentran organizados de acuerdo a la consulta que los generó, de tal manera que el agente Central pueda mostrar al usuario únicamente aquellos documentos que corresponden a la consulta cuya solución solicita el usuario en un momento dado.
- Agentes subordinados : Constituye en conjunto de referencias a los apoderados (proxies) de los agentes Intermediarios que se encuentran al servicio del agente Central. Esta colección de referencias permite al agente enviar fácilmente mensajes punto a

punto a cada uno de los agentes Intermediarios, así como administrar y controlar la cantidad de agentes a su servicio.

- Propiedades del agente : Es un conjunto de parámetros configurables que determinan diversos aspectos sobre el funcionamiento del agente, cuyo propósito es guardar tales parámetros en un medio de almacenamiento secundario para preservar tal información aún después de periodos de inactividad del sistema.

3.2.7.2. Agente Intermediario

Al igual que en el caso del agente Central, las acciones del agente Intermediario tienen su origen en eventos provenientes del ambiente exterior, constituido por el usuario, otros agentes, y el sistema.

Los estímulos que tienen su origen en acciones del usuario, se transmiten al agente a través de la interfaz gráfica, constituida por una ventana. Normalmente la ventana de un agente Intermediario no es visible, sin embargo el agente Intermediario puede mostrarla si el agente Central se lo pide.

La siguiente tabla muestra las acciones del usuario sobre la ventana y las acciones correspondientes del agente Intermediario :

Evento	**Acciones**
Llamar a un	Notifica al agente Descubridor en cuestión

agente	que el usuario desea ver su ventana.
Cerrar	Cierra la ventana. Notifica a los agentes Descubridores a su servicio que está cerrando su ventana.

Interfaz de usuario del agente Intermediario :

En la siguiente figura podemos ver la ventana del agente Intermediario :

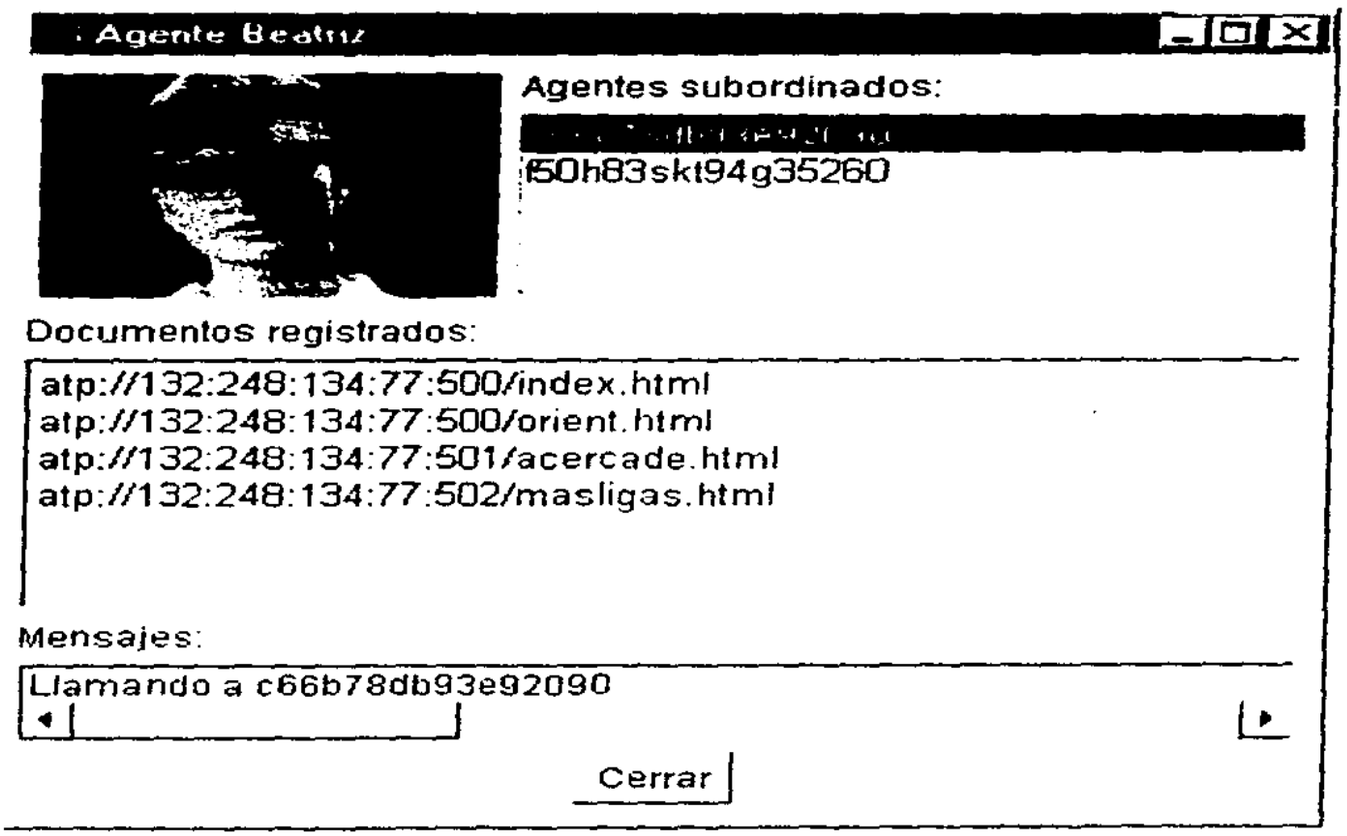


figura 3.12.

La tabla a continuación relaciona los mensajes recibidos de otros agentes con las acciones tomadas por el agente Intermediario :

Mensaje	**Acciones**
evaluatedDocument	Si el documento tiene una calificación satisfactoria, lo incorpora a su base de documentos registrados, y modifica su modelo de preferencias del usuario en base a información extraída del documento.
solveQuery	Selecciona de su base interna de documentos aquellos que satisfagan la consulta. Envía copias de tales documentos al agente Central.
showWindow	Muestra su ventana en pantalla
newDocuments	Selecciona los documentos que cumplan con las preferencias actuales del usuario. Envía el resto a otros agentes Intermediarios Envía los documentos seleccionados al agente Central para mostrarlos al usuario y obtener su evaluación. Envía las preferencias actuales del usuario al agente Descubridor.

rejectedDocuments	Selecciona los documentos que le interesen. Suma tales documentos a su base interna de documentos. Envía documentos al agente Central para su evaluación.
terminating	Salva las propiedades y preferencias del usuario en disco. Cierra su ventana si está abierta.

Los eventos originados en el servidor de aglets (Tahiti aglets server) también causan una reacción por parte del agente Intermediario. La siguiente tabla muestra la relación entre tales eventos y las acciones del agente :

Evento	**Acciones**
Create	Carga las propiedades y preferencias del usuario en memoria. Se subscribe a ciertos mensajes que son difundidos por otros agentes. Crea su ventana. Crea los agentes Descubridores necesarios.
Dispatch	Arroja una excepción de seguridad que será manejada por el aglets server (Tahiti)
Revert	Arroja una excepción de seguridad que

	será manejada por el aglets server (Tahiti)
Deactivate	Oculta la ventana
Activate	Muestra o trae al frente la ventana
Dispose	Salva las propiedades y preferencias del usuario en el disco. Cierra la ventana. Notifica a todos los agentes Descubridores que se encuentren presentes que está cerrando su ventana.

Estructura funcional del agente Intermediario :

El siguiente diagrama muestra la estructura funcional del agente Intermediario :

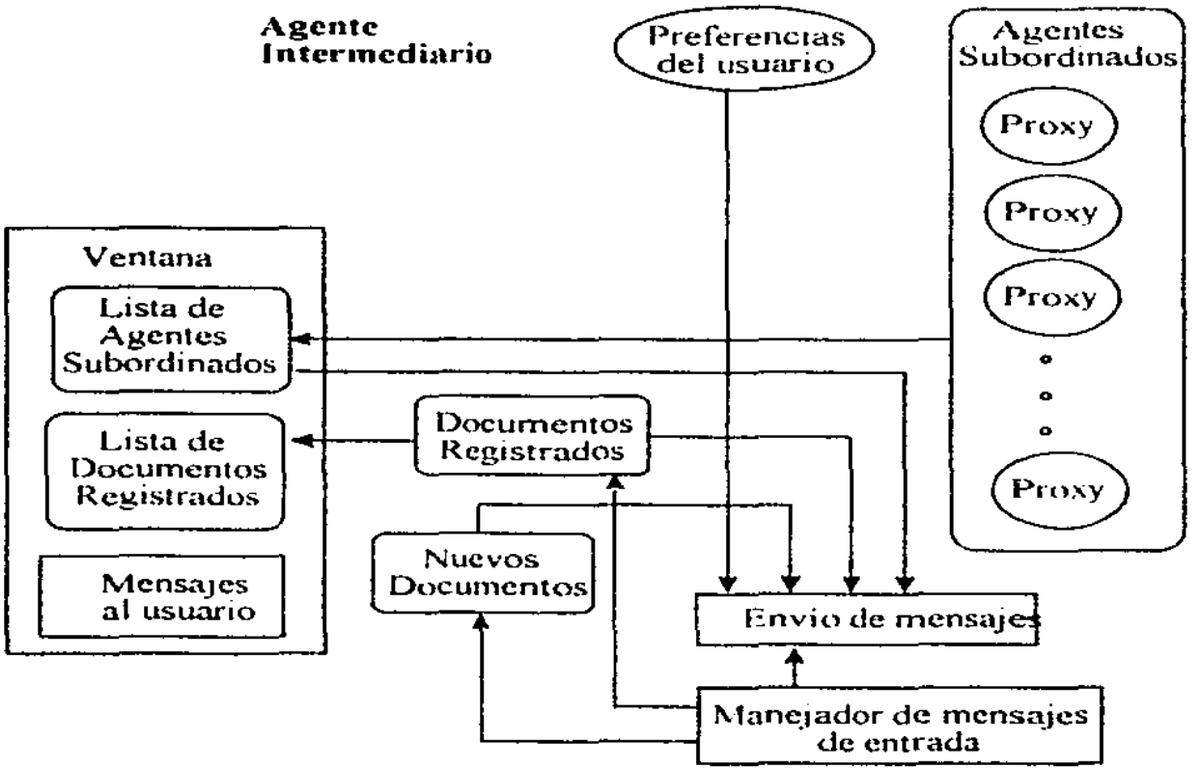


figura 3.13.

3.2.7.3. Agente Descubridor

El agente Descubridor es el único de los tres tipos de agentes que intervienen en este sistema, que tiene que habitar, no en uno, sino en una gran cantidad de ambientes diferentes, cada uno de ellos de naturaleza altamente complicada. Por tal motivo, los mecanismos que regulan sus actividades requieren de una mayor complejidad.

Podemos caracterizar el ambiente que habitará el agente Descubridor como :

- Inaccesible. Existen atributos del ambiente que intervienen directamente en el éxito del Agente en la realización de sus tareas, y que sin embargo no pueden ser conocidos por el aparato sensor del agente.
- Dinámico. El ambiente cambia a través del tiempo, y es posible que lo haga mientras el agente se encuentra decidiendo su próxima acción
- No determinístico . Los cambios que se sucedan en el ambiente no pueden ser determinados a partir de su estado actual.

Además de las dificultades arriba mencionadas, existe otra característica de éste agente que justifica la mayor elaboración de su sistema de toma de decisiones, y que constituye la principal diferencia entre el agente Descubridor y los otros dos tipos de agentes. Es su proactividad.

A diferencia de los agentes Central e Intermediario, el agente Descubridor no actúa únicamente incitado por estímulos externos, sino que lo hace también con el deseo de cumplir un conjunto de metas que le han sido asignadas desde su creación. Esa es la razón de que se encuentre siempre en estado de ejecución, mismo que sólo se interrumpe para atender a los eventos externos originados por

mensajes, eventos en la interfaz de usuario, o eventos originados en el servidor de agentes.

En los dos tipos de agentes mas simples, el sistema de decisión consta únicamente de un conjunto de reglas condición-acción, en tanto que en el agente Descubridor, el sistema está conformado de varios procesos y repositorios de información que trabajan conjuntamente para proveer al agente de un comportamiento mas flexible (ver figura 3.14).

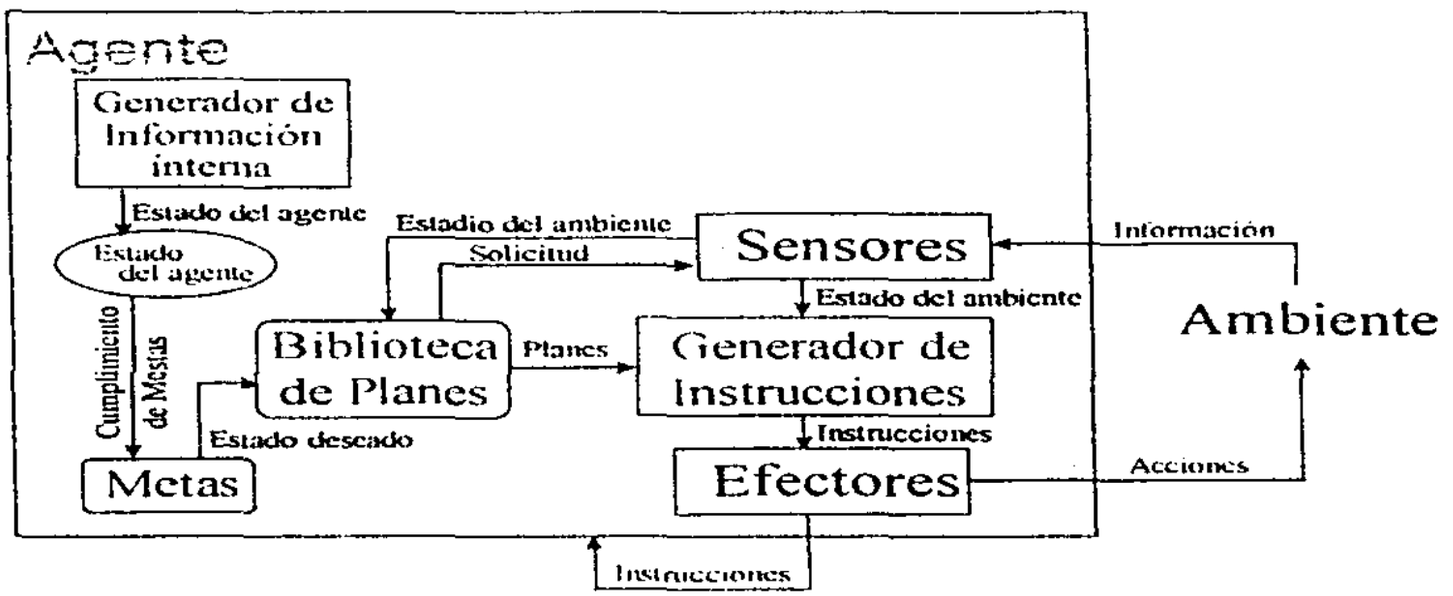


figura 3.14.

Esta arquitectura sigue los lineamientos del sistema de razonamiento procedural o PRS [30], propuesto por Georgeff e Ingrand, en 1989, que permite razonar acerca de las acciones que deben realizarse en

ambientes dinámicos. En este sistema, el estado del ambiente, el estado del agente, el estado deseado, y las intenciones, se representan explícitamente y juntos determinan las acciones del sistema en una situación determinada.

Básicamente, la principal mejora consiste en que el Generador de Instrucciones no genera por si mismo las acciones que los efectores deberán llevar a cabo, sino que utiliza para ello un plan facilitado por una biblioteca de planes.

Un plan es un conjunto de acciones que pretenden realizarse de manera secuencial y ordenada. La biblioteca de planes es capaz de sugerir un plan si conoce cuál es el objetivo del agente, es decir, a dónde quiere llegar. Esa meta que el agente desea lograr está representada por un estado.

Un estado es una colección de atributos del agente o del ambiente, asociados cada uno de ellos con un valor. De esta manera tenemos que una meta no es mas que un estado deseado por el agente.

Debido a que el ambiente en que existirá el agente Descubridor no puede ser alterado por éste (y no debe, por cuestiones de seguridad), las metas del agente quedan expresadas en términos de un estado deseado del agente en sí, y no en términos de un estado deseado del ambiente, que no puede ser manipulado (la única acción que un agente puede realizar sobre el ambiente es el envío de mensajes a otros

agentes, pero eso no se toma en consideración por dos razones : el agente no conoce el efecto de sus mensajes en los demás agentes, y, la representación interna del ambiente, y del estado del agente, no incluye una representación del estado de los demás agentes, por lo tanto, tal representación no puede ser inluida como una meta).

Si la biblioteca de planes encuentra que existe mas de un camino para lograr el estado deseado, decide pedir información relevante acerca del estado actual del ambiente, es decir, factores ambientales de cuyos valores dependa la convenciencia de aplicar uno u otro de los posibles planes.

La biblioteca de planes tiene la siguiente forma:

Status actual	Status deseado	Plan
Status	Status	Acción 1. Acción 2. . .Acción n
Status	Status	Acción 1. Acción 2. . .Acción n
:	:	:
Status	Status	Acción 1. Acción 2. . .Acción n

El agente, por otra parte, tiene solo una meta que lograr, que es obtener un reconocimiento por parte del agente Intermediario que lo creó. Sin embargo, toda meta puede estar constituida por un conjunto de submetas, cada una mas sencilla de lograr que la meta superior, que a su vez es fácil de alcanzar cuando se ha logrado cada una de las submetas de que consta. De esta manera, la meta del agente es en

realidad un árbol de metas y submetas, y el agente trata siempre de satisfacer las metas mas básicas.

El árbol de metas tiene la siguiente forma :

Meta

Nombre			
Estado que la representa			
¿Ha sido lograda?			
Meta 1	Meta 2	...	Meta n

El agente realiza una revisión periódica del estado actual para identificar las metas que quedan satisfechas con ese estado, de manera que pueda proseguir intentando alcanzar las metas subsecuentes.

Los sensores del agente están constituidos por los siguientes mecanismos :

- La interfaz con el sistema remoto
- El manejador de mensajes
- La ventana de usuario
- Las interfaces de movilidad y persistencia.

La interfaz con el sistema remoto permite al agente obtener información de ese sistema de manera local, es decir, una vez que el agente ha viajado hasta allí y el sistema ha aceptado hospedarlo. Ésta información puede ir desde el nombre del host hasta información sobre la red a la que pertenece. Queda como responsabilidad del diseñador de los agentes decidir la cantidad, calidad y variedad de la información a la que el agente tendrá acceso estando en un host remoto.

A pesar de que el agente Descubridor es un agente orientado a metas, es también capaz de reaccionar ante eventos de la manera que se describe a continuación :

El manejador de mensajes indica al agente cuando éste ha recibido un nuevo mensaje proveniente de otro agente, en tanto que la ventana de usuario notifica al agente cuando un evento ha sido originado por éste.

Las interfaces de mobilidad y persistencia indican al agente cuando un evento de creación, desctrucción, movilidad, activación o desactivación, ha sido originado en el servidor de agentes.

Los efectores, por otra parte, están constituidos por métodos que pueden realizar acciones que tienen su efecto ya sea en el ambiente (envío de mensajes), o en el propio agente y sus componentes internos.

En la figura 3.15 se muestra la interfaz de usuario del agente Descubridor

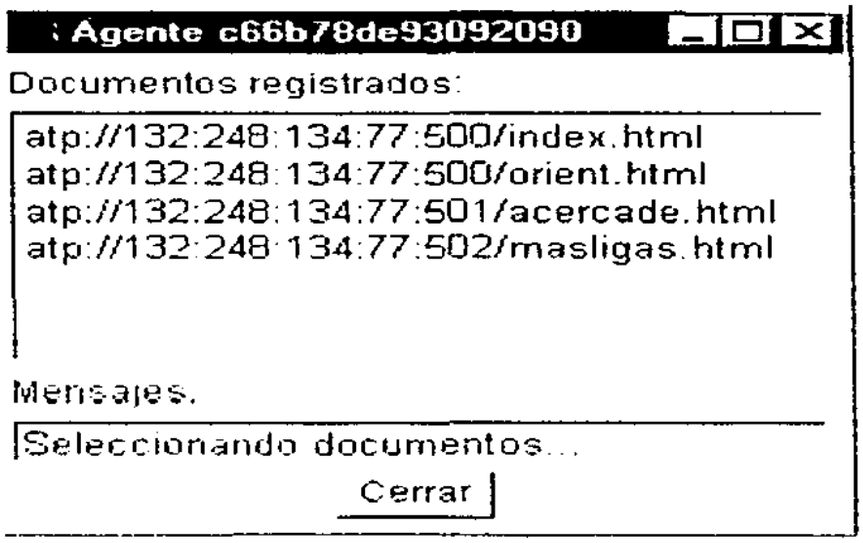


figura 3.15.

El siguiente diagrama esquemático muestra el flujo de datos en un agente Descubridor :

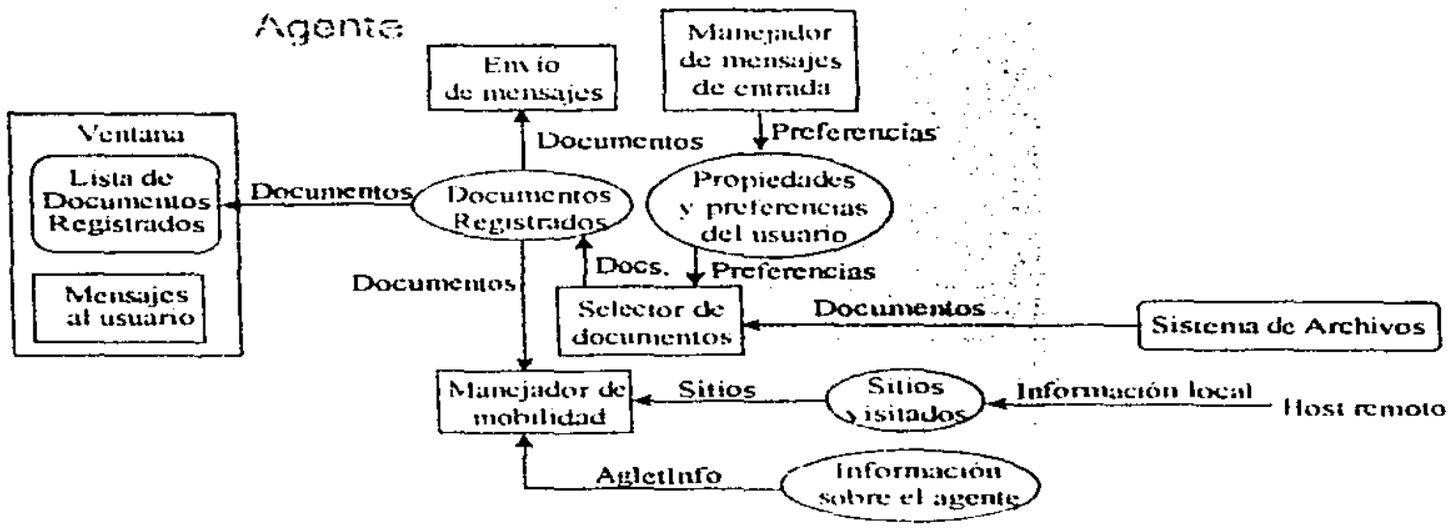


figura 3.16.

Durante la creación de un nuevo agente Descubridor, el agente Intermediario al cual servirá, le facilita las preferencias del usuario a las cuales deberá apegarse, y lo vuelve a hacer cada vez que el agente Descubridor regresa a la computadora hogar, de manera que éste último cuente siempre con una representación actualizada de las preferencias del usuario.

Cuando el agente se encuentra ejecutándose en un host remoto, accesa el sistema de archivos y utiliza tales preferencias para seleccionar los documentos que considere de interés para el usuario. El nombre y la localización de los documentos, así como una descripción de su contenido, son almacenados en un repositorio interno de documentos, y mostrados en la ventana del agente.

Además, el agente cuenta con una lista interna de los sitios que ya ha visitado, y utiliza dicha información, junto con la lista de documentos e información sobre el agente mismo, para decidir cuáles serán sus próximos movimientos por la red.

3.2.8. Adaptación y aprendizaje

De todos los aspectos que abarca el comportamiento del sistema multiagentes que nos ocupa, sólo dos son susceptibles de cambiar a lo largo del tiempo, con el fin de ofrecer al sistema facilidad de adaptación a las necesidades cambiantes del usuario y a la naturaleza dinámica de la red. Estos dos factores son : el número de agentes del que dispone el sistema, y el tema al que se orienta cada uno de ellos.

El número de Intermediarios en el sistema responde a la variedad de los temas en los que el usuario muestra interés, y corresponde al agente Central controlar su número, de manera que se ofrezca al usuario un servicio que cubra todas las áreas de interés, y que no se ocupe de temas en los que no lo hay.

La tarea correspondiente del agente Intermediario es controlar el número de agentes Descubridores que se encuentran a su servicio, de tal manera que la cantidad de documentos que periódicamente entrega al agente Central como sugerencias, sea conveniente.

Por otro lado, es también el agente Intermediario el responsable de mantener actualizada su representación interna de las preferencias del usuario respecto a el tema particular que maneja, analizando la manera como el usuario evalúa los documentos sugeridos, y registrando los posibles cambios en tal representación.

En un principio, el sistema cuenta solo con el agente Central. Cuando el usuario realiza una consulta explícita al agente Central, éste a su vez, hace la misma pregunta a cada uno de los agentes Intermediarios para que trate de ofrecer documentos adecuados. En este caso, el agente Central no cuenta con ningún agente Intermediario que responda a la pregunta, por lo que decide crear uno nuevo, encargado exclusivamente de responder a la consulta.

Una consulta se considera como un conjunto de características que se espera que contenga un documento ofrecido como respuesta. En este sistema, la única característica que compone a una consulta, es una serie de palabras clave. Esta característica es por lo tanto, la misma que se utiliza para obtener la caracterización de un documento.

Las palabras clave constituyen la manera en que los agentes registran las preferencias del usuario, las consultas hechas por él, y el contenido de los documentos. Sin embargo, es relativamente fácil proveer a los agentes de una metodología mas completa de caracterizar tales aspectos, tarea que queda fuera del alcance de este trabajo.

Una palabra clave consta de dos partes : la palabra en sí, y un campo numérico que indica, según el caso, la frecuencia relativa con que aparece en un documento, o la importancia relativa que tiene dentro de conjunto de palabras que describen las preferencias del usuario. As tanto los documentos como las preferencias del usuario respecto cierto tema, quedan representados en un conjunto de palabras clave (ver figura 3.17) [15].

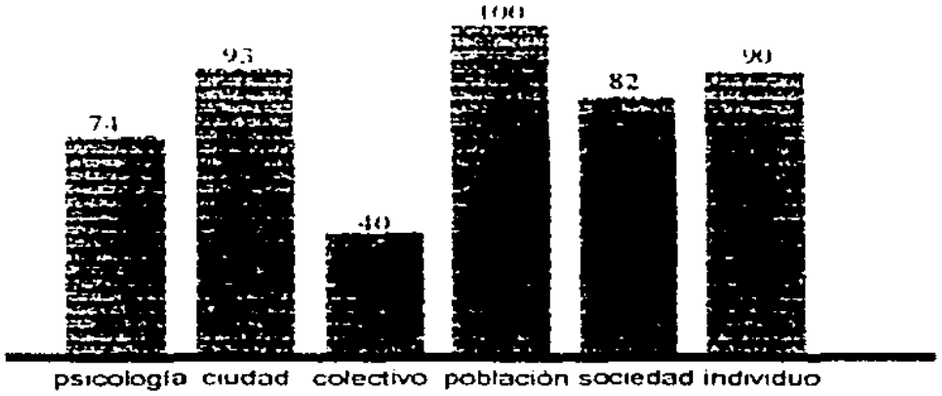


figura 3.17.

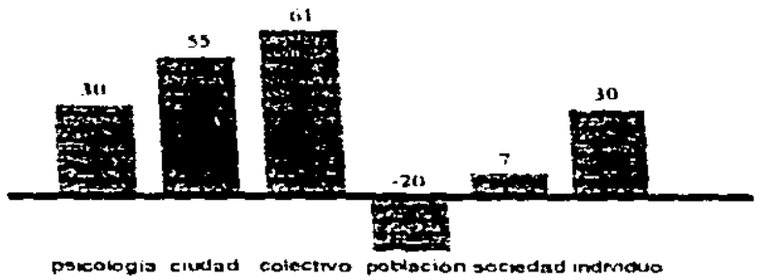
Cuando, con el propósito de responder a una consulta, un agent Intermediario es creado, las palabras clave que integran la consult pasan a ser la representación de las preferencias del usuario para es agente, con porcentajes de interés del cien por ciento para cada una d ellas. A su vez, cuando el agente Intermediario crea los agente Descubridores que le servirán, les facilita tal información, de modo qu

puedan seleccionar los documentos adecuados cuando viajen a otros servidores.

Para seleccionar los documentos que satisfacen las preferencias del usuario, es necesario antes, obtener una representación análoga del documento, para entonces comparar ambas y decidir si el documento puede ser de interés.

El agente es capaz de leer documentos html (Hipertext Markup Language) del sistema remoto en donde se encuentra, y obtener las frecuencias de ocurrencia de cada una de las palabras que cada uno de ellos contiene, elaborando un histograma para cada documento. Posteriormente selecciona las palabras clave mas significativas (las que sobrepasen el 60% de ocurrencia), y elabora con ellas un nuevo histograma normalizado.

Un conjunto de palabras clave está normalizado cuando la frecuencia mas alta es del 100%, y la mas baja es mayor o igual a cero. La figura 3.18. muestra un ejemplo de normalización.



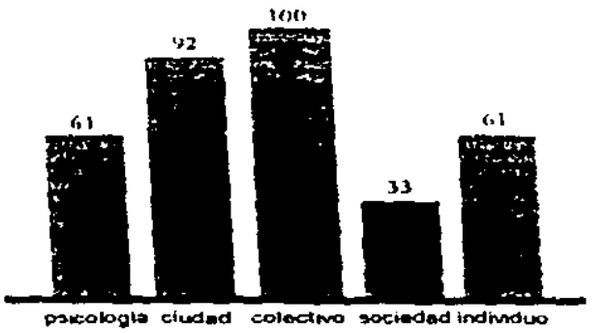


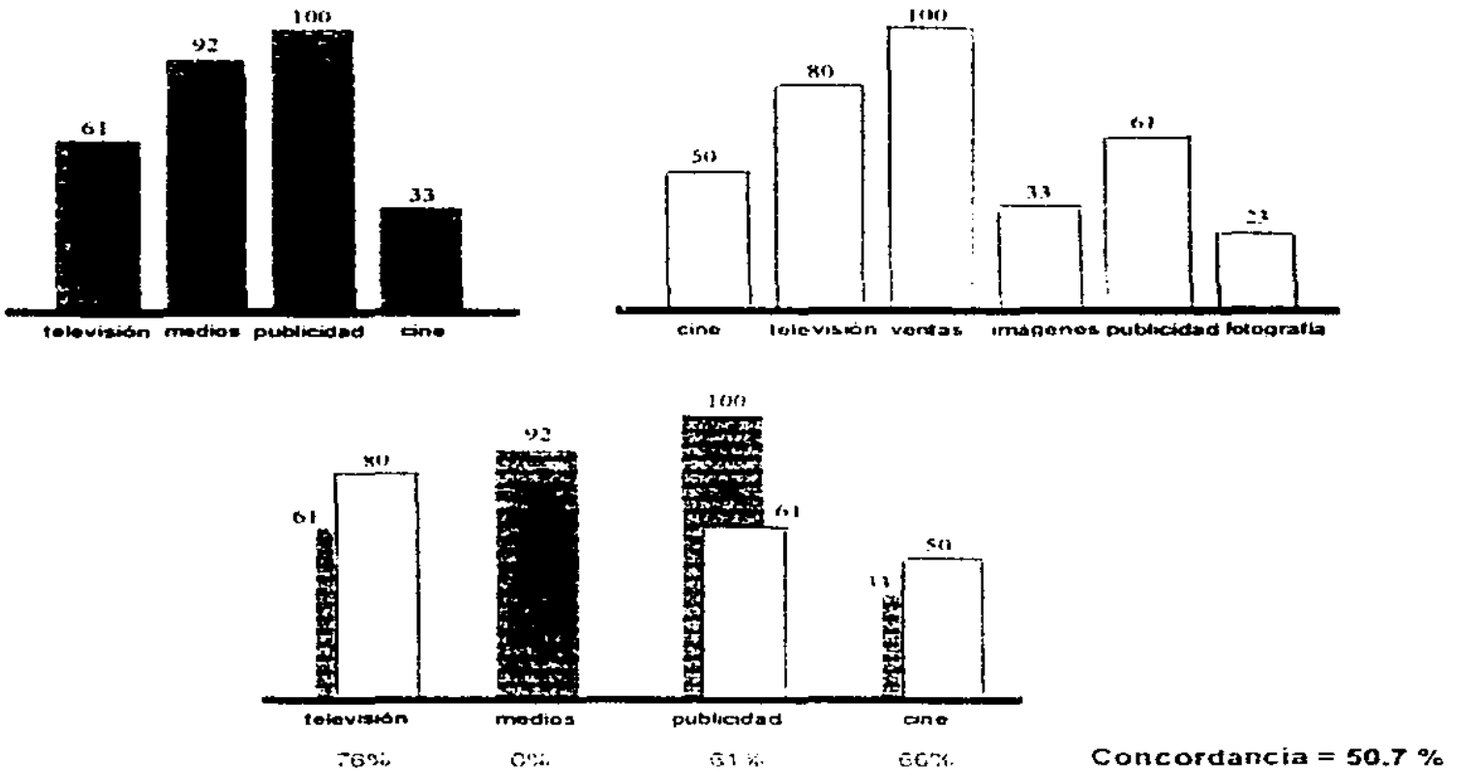
figura 3.18.

Es necesario aclarar que las frecuencias negativas no aparecen como resultado de la caracterización de un documento, sino como resultado de el ajuste de las preferencias del usuario, como veremos mas adelante. (Nota : las palabras que después de la normalización no reúnen un mínimo de tres por ciento, son eliminadas del vector) .

Cuando se ha leído un documento, el conjunto de palabras clave que lo caracterizan se compara con el conjunto de palabras que caracterizan las preferencias del usuario, proceso que se realiza de la siguiente manera (ver figura 3.19):

- Se determina la relación geométrica entre cada una de las palabras que aparezca en ambos conjuntos, sin importar en cuál de los dos aparezca con una frecuencia mayor.
- Se promedian tales porcentajes respecto al número de palabras que describen al usuario (ya que por definición es ése el perfil que se trata de satisfacer completamente).

- Si el promedio sobrepasa el 40% de concordancia, el documento es aceptado.



Concordancia = 50.7 %

figura 3.19.

Cuando el agente Descubridor en cuestión llega al sitio de donde partió, entrega al agente Intermediario los documentos conseguidos. No obstante, es posible que durante su ausencia el perfil del usuario haya cambiado, por lo que el agente Intermediario realiza una segunda selección al conjunto de documentos utilizando el nuevo perfil como criterio. Si el agente considera que algunos documentos recién traidos ya no son del interés del usuario, los envía a cada uno de los demás

agentes Intermediarios en el sistema, ya que cabe la posibilidad de que entren dentro del área de trabajo de alguno de ellos. Finalmente, el agente se cerciora de que ninguno de los documentos nuevos haya sido traido ya en alguna ocasión anterior, si esto sucede, el documento es destruido

Ahora, ¿cómo se mantiene actualizada la representación de las preferencias del usuario ? El agente Intermediario, envía frecuentemente, según el ritmo de lectura del usuario, nuevos documentos al agente Central con el fin de que éste los muestre al usuario. El usuario debe leer tales documentos y calificarlos, de acuerdo a qué tanto se acercan a sus intereses, en una escala del -4 al 4.

Una vez calificado un documento, éste es enviado de vuelta, junto con la calificación asignada, al agente Intermediario responsable de su obtención. Éste a su vez decide si conservar el documento para responder futuras preguntas (si la calificación es mayor a cero), y además modifica el perfil del usuario de acuerdo a tal documento y su calificación

Esta modificación se realiza de la siguiente manera (figura 3.20):

- Se pondera el vector de palabras clave del documento de acuerdo a su calificación (teniendo al 5 como cien por ciento y al -5 como menos cien por ciento)

- Se suma el vector resultante al vector de palabras clave que define el perfil de usuario.
- Se normaliza el vector que define el perfil de usuario.

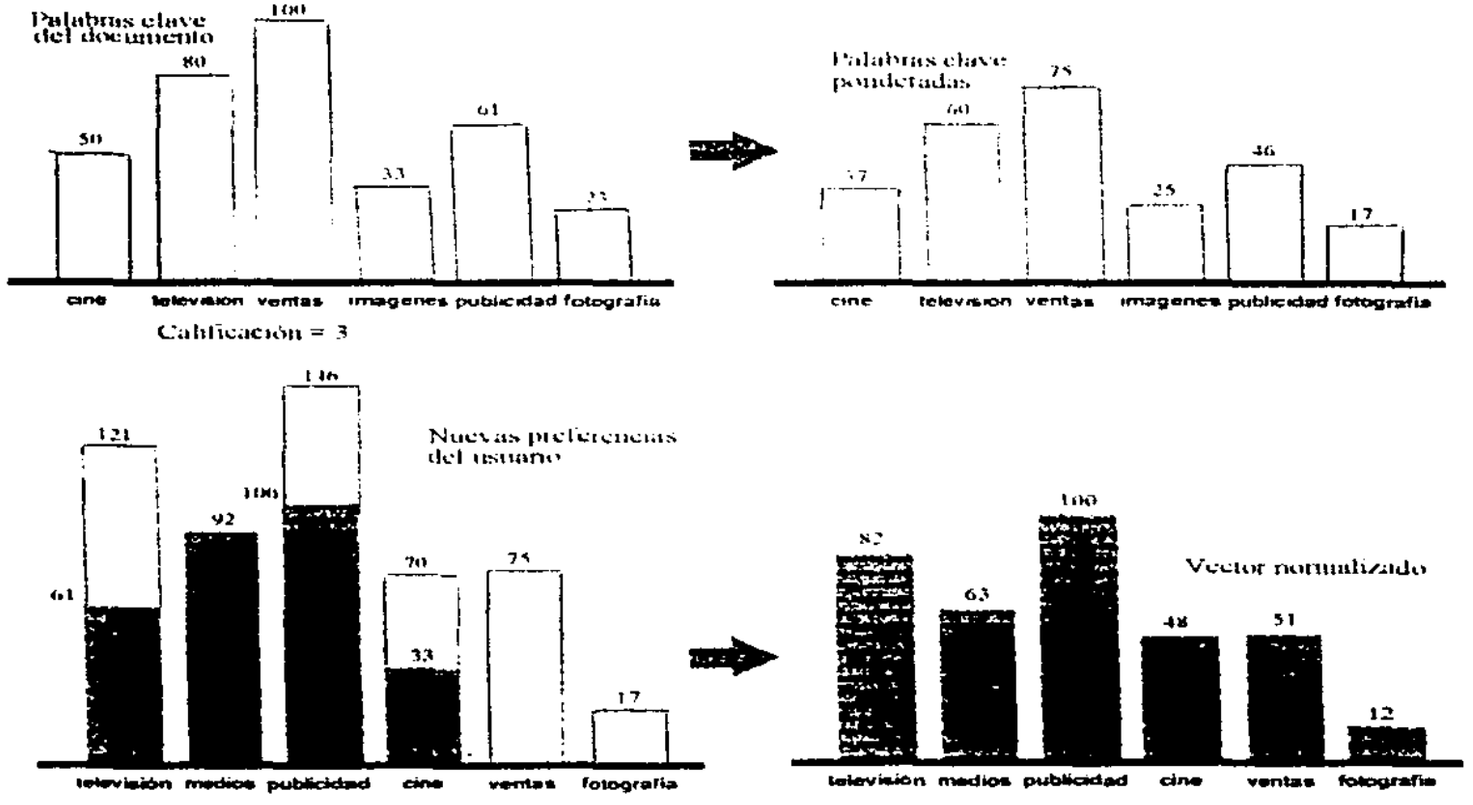


figura 3.20.

De esta manera se consigue que cada agente Intermediario posea una representación actualizada de las preferencias del usuario respecto al tema del que se ocupa.

Periódicamente, el agente Intermediario revisa el conjunto de documentos que ha registrado y determina cuáles documentos aún corresponden al perfil que define al usuario, eliminando aquellos que ya no lo hagan. Esta base actualizada de documentos es utilizada por el agente para tratar de responder a consultas directas del usuario, que en algún momento quiere que el sistema le recuerde cuáles documentos hablan de algún tema.

3.3. Explicación de los programas

El objetivo de esta sección es dar a conocer la manera en que la funcionalidad descrita en la sección anterior es lograda mediante la programación, así como identificar los aspectos del sistema que por no ser el punto central de este trabajo, no han sido implementados en toda su amplitud y con todas las consideraciones que merecen.

Esta sección pretende también ser una referencia técnica que facilite el análisis del sistema y permita una eventual implementación mas amplia y completa. Consulte los apéndices A y B si desea revisar el código fuente del sistema.

3.3.1. Limitaciones y oportunidades de mejora

La funcionalidad de nuestro sistema multiagentes es lograda mediante la interacción de una gran variedad de tecnologías que conforman diferentes aspectos de su implementación. Por tal motivo,

estudiaremos las limitaciones del sistema dividiendo esta sección en dos partes. Podemos identificar las siguientes facetas :

El manejo de errores y excepciones

Los errores y situaciones excepcionales en el sistema multiagentes no se encuentra completamente cubierto, es decir, toda excepción cuenta con una instrucción *catch* que previene que el sistema multiagentes deje de funcionar, pero muy pocas instrucciones *catch* han sido implementadas para tomar acciones correctivas. Por otra parte, el sistema no cuenta con un conjunto propio de excepciones, sino que emplea únicamente las proveídas por el lenguaje Java [5].

La seguridad

La seguridad en un sistema multiagentes es algo que no puede ser ignorado. Después de todo, una máquina anfitriona de agentes está permitiendo que un objeto activo (el agente) corra dentro de su espacio de direccionamiento. (Un virus informático es escencialmente un agente móvil mailicioso.) Ningún sistema comercial basado en agentes debe ser desarrollado sin que posea mecanismo de seguridad robusto.

Debe establecerse cierto grado de confianza entre el agente y el host. Un host puede ser capaz de reconocer la identidad de un agente y entonces establecer las restricciones apropiadas en cuanto a las acciones permitidas al agente, basándose en sus privilegios de acceso.

De igual manera un agente debe asegurarse de que un host es realmente el host que desea visitar, y no una versión falsa esperando a capturar el agente para aprovechar la información confidencial que éste pueda contener.

Cierto grado de seguridad es ampliamente provisto por la arquitectura intrínseca de de Java, y en las características adicionales de seguridad del JDK, pero al igual que en el caso de los applets, algunos ataques, como reservar memoria hasta que la máquina falle, aún son posibles.

Actualmente, Tahiti, el servidor de aglets, impone restricciones de seguridad muy severas a las actividades de cualquier aglet que no se haya originado localmente. Por ejemplo, un agente no puede llamar a librerias dinámicamente a menos que el servidor en el que se encuentra haya sido configurado con la opción -nosecurity (sin seguridad).

Tahiti define por el momento dos categorías de agentes en lo que a confianza se refiere [6]:

- Trusted : Son agentes que se han originado localmente utilizando el comando "Create Aglet" del panel. Un agente "trusted" tiene el privilegio de crear otro agente de la misma categoría.
- Untrusted : Son los agentes que proviene de un servidor remoto o que es creado como una instancia de una clase localizada en un sitio

remoto. En general, el contro de acceso a los recursos debe especificarse de manera mas estricta para un agente "untrusted".

El control de acceso a recursos se define respecto a cuatro categorias :

- Control de acceso a archivos: Especifica los directorios del sistema de archivos a los que los agentes tendrán acceso.
- Control de acceso a la red : Especifica los puertos a los que los agentes pueden escuchar y a los cuales pueden conectarse.
- Propiedades.
- Otros: En este panel puede especificarse el acceso a : Mensajes de advertencia, abrir ventanas, JDBC (Java Data Base Conectivity), Cliente RMI, y Servidor RMI.

Aunque actualmente Tahiti define solo dos categorías de agentes, se planea extender este modelo a un mecanismo de seguridad basado en capacidades, para brindar un control mas flexible sobre la seguridad.

En la implementación de nuestro sistema multiagentes, para propósitos de desarrollo y demostración, se han asignado las mismas facilidades de acceso a recursos a ambos tipos de agentes, es decir, todos los permisos necesarios sin importar la categoria de confianza en que se encuentren.

La serialización de objetos que el lenguaje Java provee es un proceso muy lento, que ocasiona grandes tiempos de espera cuando un agente se dispone a viajar de un servidor a otro. Optimizar y reducir el tamaño de los agentes y de los objetos que estos llevan consigo permitiría alcanzar mayores velocidades de migración y por lo tanto una mayor eficiencia del sistema.

3.3.2. Jerarquía de Clases

El diagrama en la figura 3.21 muestra la herencia de las clases que componen el sistema multiagentes.

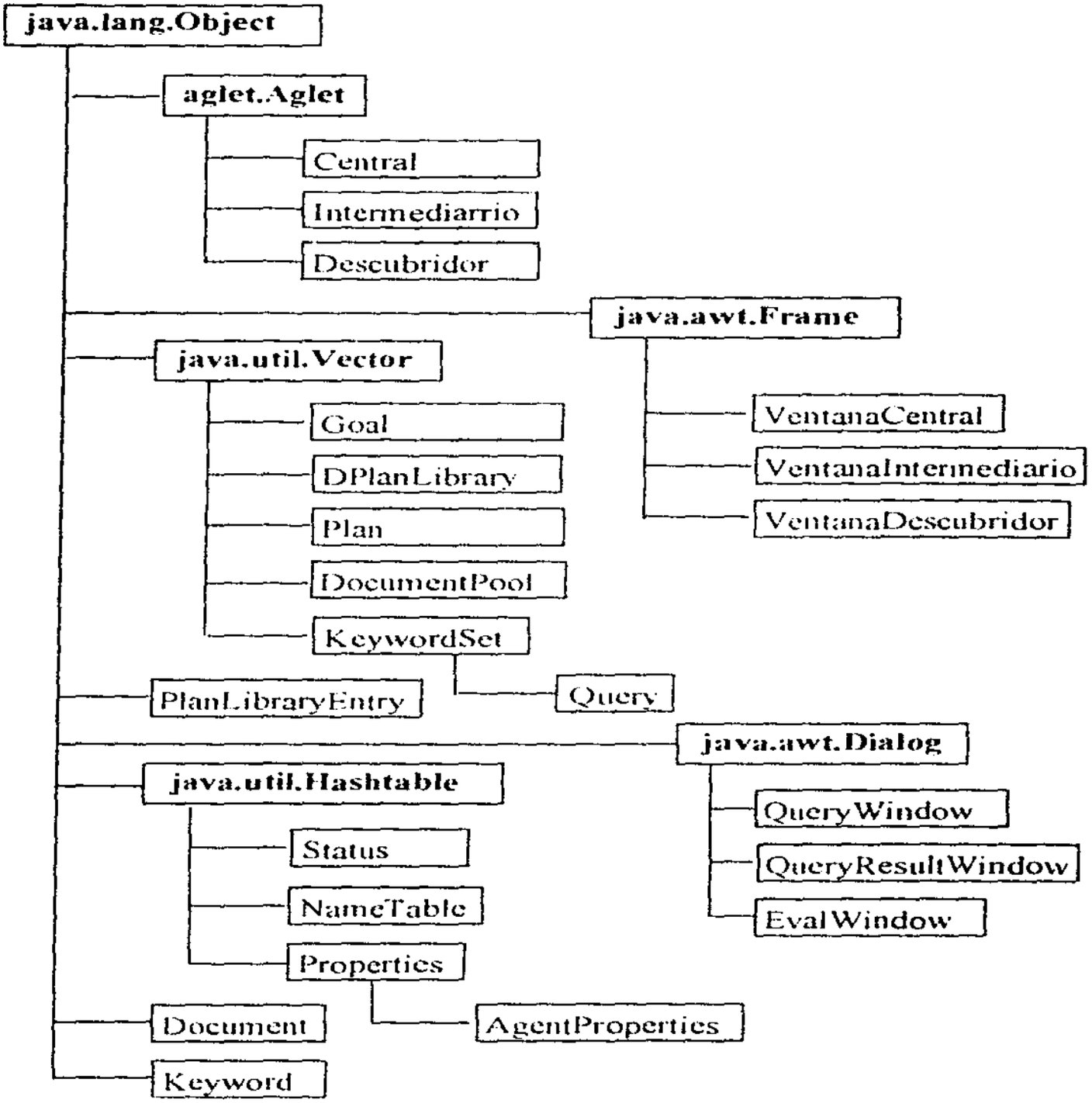

java.lang.Object
aglet.Aglet
Central
Internediarrio
Descubridor
java.awt.Frame
VentanaCentral
VentanaInternediario
VentanaDescubridor
java.util.Vector
Goal
DPlanLibrary
Plan
DocumentPool
KeywordSet
Query
PlanLibraryEntry
java.awt.Dialog
QueryWindow
QueryResultWindow
EvalWindow
java.util.Hashtable
Status
NameTable
Properties
AgentProperties
Document
Keyword

figura 3.21.

3.3.3. Jerarquía de paquetes

Los paquetes son la herramienta que brinda Java para organizar y diseñar aplicaciones a gran escala (figura 3.22). Se utilizan para categorizar y agrupar clases. El paquete al que pertenece cada clase aparece en el encabezado del archivo java correspondiente a esa clase

Es necesario hacer notar que debe haber una correspondencia entre la organización jerárquica de los paquetes y la manera en que las clases se encuentran almacenadas en los directorios del sistema Por ejemplo, las clases que pertenecen al paquete agentes.conocimiento, deben estar guardadas en el directorio /agentes/conocimiento. Todas las clases utilizadas en este sistema son públicas, es decir, que pueden ser accesadas por clases que pertenezcan a cualquier otro paquete.

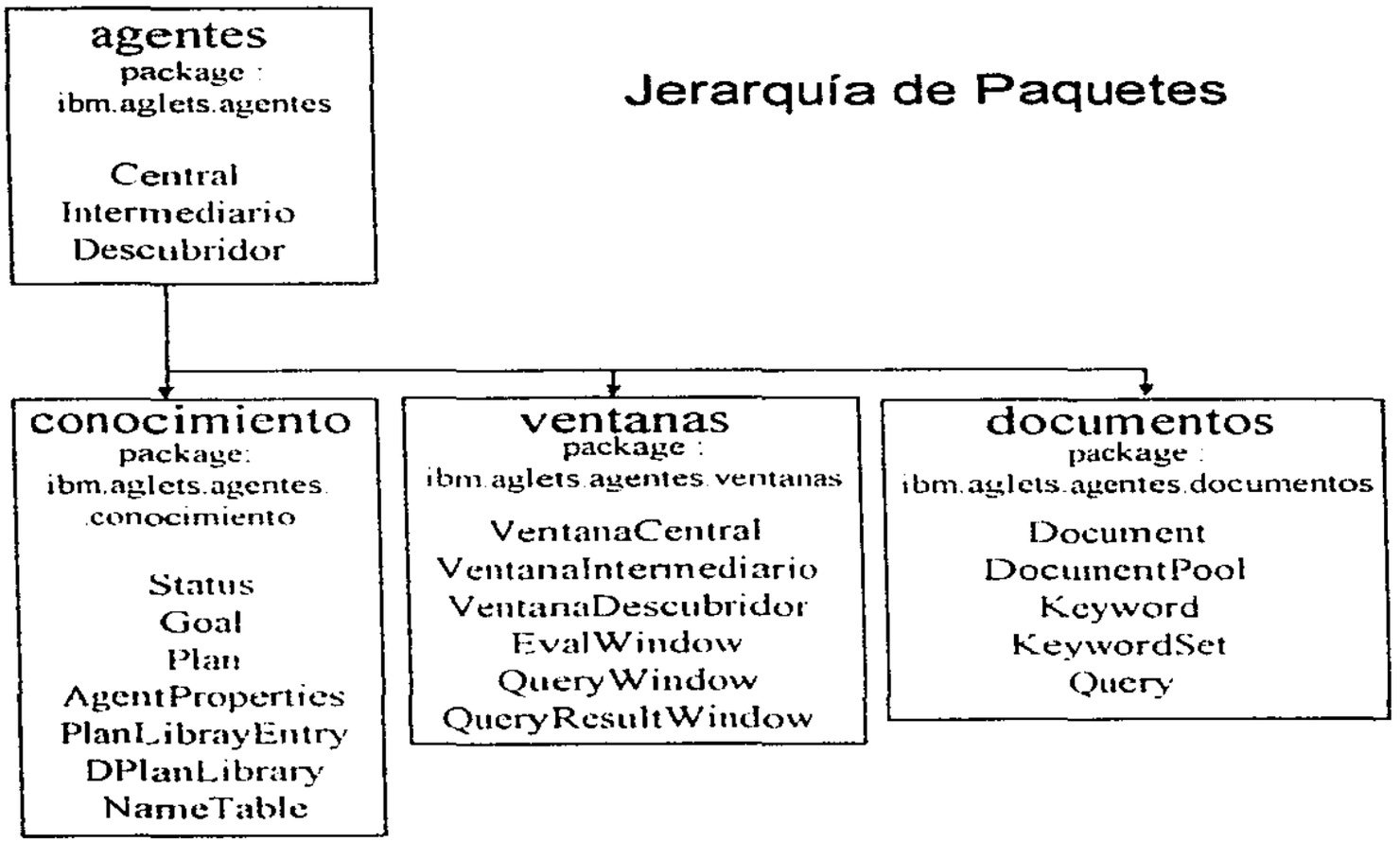

agentes
package :
ibm.aglets.agentes
Central
Intermediario
Descubridor
Jerarquía de Paquetes
conocimiento
package:
ibm.aglets.agentes.
conocimiento
Status
Goal
Plan
AgentProperties
PlanLibrayEntry
DPlanLibrary
NameTable
ventanas
package :
ibm.aglets.agentes.ventanas
VentanaCentral
VentanaInternediario
VentanaDescubridor
EvalWindow
QueryWindow
QueryResultWindow
documentos
package :
ibm.aglets.agentes.documentos
Document
DocumentPool
Keyword
KeywordSet
Query

figura 3.22.

Parte IV, Resultados.

En esta cuarta parte se expondrá el desempeño obtenido con el sistema multiagentes, y se medirá tal desempeño en términos de varios parámetros de interés. Se harán funcionar los varios servidores de agentes que compondrán la ruta de cada agente.

4.1. Parámetros evaluados

Los parámetros a evaluar son de tres tipos distintos. Primeramente se medirán los tiempos que tardan en efectuarse las operaciones mas frecuentes e importantes que tienen lugar en el sistema multiagentes. Segundo, se evaluará la cantidad de documentos obtenida mediante el sistema por unidad de tiempo y en relación con la cantidad que se encuentra disponible en la red (en este caso un número limitado de servidores), así como el tiempo que tarda el sistema en responder a una consulta. Por último se evaluará la capacidad de ajuste tanto de cada agente como del sistema completo, ante un cambio en el perfil de usuario. Así, los parámetros evaluados son :

- Tiempo de despacho de un agente
- Tiempo de creación de un agente
- Tiempo de respuesta a una consulta con antecedentes
- Cantidad de documentos obtenidos
- Evaluaciones necesarias para el ajuste del perfil de cada agente.

- Evaluaciones necesarias para el ajuste del perfil en el sistema completo.

4.2. Pruebas realizadas

Se realizaron únicamente pruebas locales, es decir, se echaron a andar varios servidores de agentes en puertos diferentes de una sola computadora. De esta manera se garantiza que las operaciones realizadas no se ven influenciadas por la subred (que involucra variables como carga, tráfico, velocidad, etc.). La computadora utilizada es una computadora personal con procesador 80486 a 66 MHz, con 20 Mbytes de memoria, y sistema operativo Windows 95.

Las mediciones de tiempo se realizaron mediante la incrustación en los programas, de puntos de señalización que indican en pantalla el momento en que inicia la operación, y el momento en que se completa. Esta técnica permite tomar la diferencia entre ambos tiempos como el tiempo de duración de la operación.

Para poder medir la cantidad de documentos obtenidos por un agente por unidad de tiempo, fue necesario realizar los siguientes ajustes :

- Bloquear el mecanismo de ajuste del perfil de usuario en el agente, de manera que la representación de las preferencias del usuario permaneciera fijo y permitiera comparar los documentos obtenidos

con el número de documentos sobre ese tema fijo que se encuentran disponibles en los diferentes sitios.

- Introducir en los servidores documentos artificiales con un grado de proximidad con las preferencias del usuario del cien por ciento, de manera que resulten elegibles.

Se introdujeron 60 documentos elegibles, distribuidos en tres sitios diferentes a razón de 20 documentos por sitio, y proveyendo al sistema de un solo agente se procedió a monitorear el número de documentos sugeridos al usuario cada cinco minutos.

Para medir el tiempo de respuesta a una consulta sin antecedentes, es decir, aquella sobre la que el sistema no había buscado información, se realizaron las siguientes medidas simplificativas :

- Se inicializó el sistema con un número nulo de agentes Intermediarios, de manera que la consulta diera origen al primero de ellos
- Se convino que la respuesta a la consulta está dado por el tiempo entre la formulación de la consulta y el despliegue del primer documento relativo obtenido.

Dado que el tiempo de obtención de un primer documento depende de una ruta elegida por el agente de manera aleatoria, la prueba se repitió diez veces y se estableció como resultado el promedio de los tiempos obtenidos en cada una de ellas.

Para determinar el número de evaluaciones de documentos (calificaciones por parte del usuario) que son necesarias para hacer que el agente cambie completamente su representación del perfil de usuario se realizaron las siguientes tareas :

- Se fijó un perfil de usuario inicial, compuesto por un conjunto de cinco palabras clave.
- Se fijó un perfil de usuario final completamente diferente, para simular el hecho de que el sistema ha cambiado súbitamente de dueño.
- Se introdujo en un solo sitio un conjunto de 50 documentos artificiales cuyo contenido varía lentamente desde aquel que satisface completamente el perfil inicial (primer documento) hasta aquel que satisface completamente el perfil final (último documento).

La finalidad de esta prueba es determinar la capacidad de ajuste con la que cuenta el sistema, y compararla con la velocidad con la que el usuario requiere que el sistema se adapte a sus necesidades.

Esta prueba fue realizada de dos maneras distintas : con un solo agente Intermediario, y con un conjunto de cinco agentes Intermediarios, con la finalidad de determinar hasta que punto la cooperación entre los agentes Intermediarios ayuda a hacer converger el perfil hacia un perfil concreto.

4.3. Resultados experimentales

Tiempo de migración de un agente

El tiempo de despacho de un agente se ve influenciado por los siguientes factores : la velocidad de serialización, la velocidad de transmisión, y de la velocidad de reconstrucción. La velocidad de serialización y de reconstrucción del agente, dependen de la capacidad de procesamiento de la computadora origen y de la computadora destino, respectivamente, en tanto que la velocidad de transmisión depende de la velocidad de la conexión, de la carga de la subred en ese momento, y del tamaño de la representación serializada del agente.

La manera mas simple de realizar esta operación es despacharlo entre dos servidores de agentes que se encuentran corriendo en una misma computadora. Utilizando la computadora personal mencionada anteriormente, se obtuvo un tiempo de migración promedio de 25 segundos, (considerando únicamente los agentes Descubridores).

Tiempo de creación de un agente

Se obtuvieron los siguientes tiempos de creación :

Agente Central : 8.4 segundos
Agente Intermediario : 5.3 segundos
Agente Descubridor : 5.4 segundos

Tiempo de respuesta a una consulta con antecedentes

Llamamos una consulta con antecedentes aquella sobre la cual el sistema ya ha provisto información anteriormente, y que por lo tanto pudiera ser contestada mediante documentos que se encuentran en la base de conocimientos de los agentes Intermediarios. Los resultados de la consulta pueden verse aún antes de que todos los agentes Intermediarios hayan respondido, sin embargo, los resultados que se presentan a continuación representan el tiempo que pasa antes de que todos los agentes respondan.

Número de agentes	**tiempo de respuesta (segundos)**
1	3.2
2	3.8
3	4.4
4	5.0
5	5.6

Cantidad de documentos

Se introdujeron 60 documentos elegibles, distribuidos en tres sitios diferentes a razón de 20 documentos por sitio, y proveyendo al sistema de un solo agente se procedió a monitorear el número de documentos

sugeridos al usuario cada cinco minutos. El experimento se realizó diez veces, obteniendo el resultado promedio que se muestra en la gráfica de la figura 4.1.

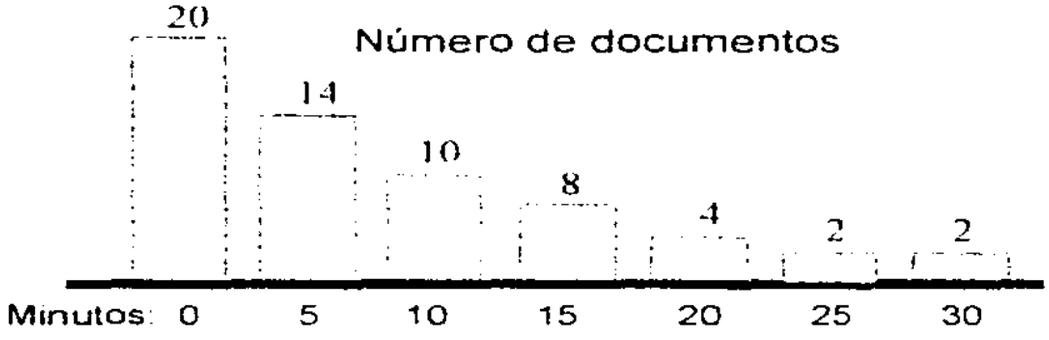


Figura 4.1.

En la gráfica el minuto cero (0) corresponde al momento en el que el agente Descubridor regresa de su primera incursión en la red, y en el cual entrega siempre veinte documentos al usuario. El tiempo que pasa entre la inicialización del sistema hasta la entrega de los primeros veinte documentos depende del tiempo que ocupa el agente en migrar de un sitio a otro (ver la sección *tiempo de migración*).

En los minutos siguientes la velocidad con la que se obtienen documentos nuevos de la red disminuye continuamente. Esto se debe a que el agente Descubridor no conserva consigo la información sobre los sitios visitados en la ronda anterior, sino que la entrega al agente Intermediario con el fin de mantenerse ligero, por lo que en ocasiones viaja a un sitio que ya había visitado antes, y trae documentos que el agente Intermediario correspondiente ya conoce.

El siguiente experimento tiene como propósito medir el tiempo promedio (medido en viajes del agente) en el que el agente termina de descubrir todos los documentos de interés que existen en la red. Los documentos se encuentran distribuidos en tres sitios diferentes. El experimento se llevó a cabo diez veces y se obtuvieron los resultados que se muestran en la figura 4.2.

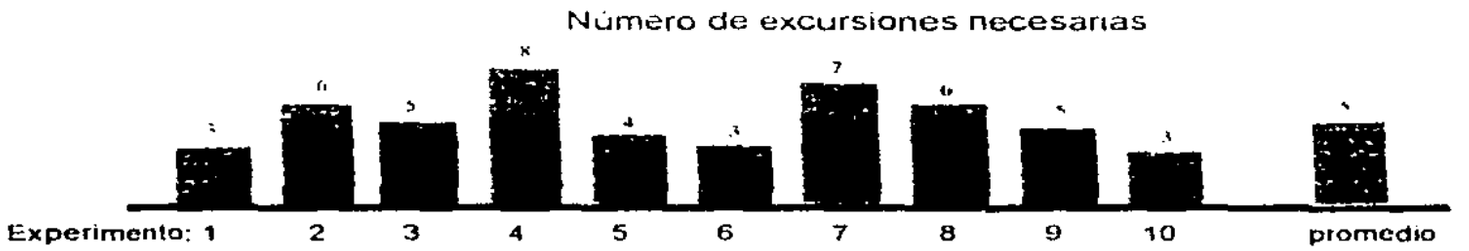


Figura 4.2

Una vez, mas, la necesidad de mantener la agilidad de los agentes Descubridores, y por lo tanto de conservar poca información dentro de ellos, obliga a que sean los agentes Intermediarios los que lleven un registro de los sitios que ya han sido visitados. Los agentes Descubridores registran los sitios que han visitado durante la búsqueda actual, sin embargo, vacian su conocimiento en los agentes Intermediarios cada vez que regresan al servidor hogar, modifican su representación de los intereses del usuario, y vuelven a partir sin ninguna información acerca de los sitios ya conocidos, por lo que es posible que un agente visite un sitio en mas de una ocasión.

Los resultados en la gráfica muestran que ésta limitación provoca que para obtener los documentos en tres servidores diferentes, es necesario que el agente realice incluso ocho incursiones en la red.

Tiempo de ajuste de perfil de cada agente (excursiones)

Se creó un agente con el siguiente perfil :

Keyword	Porcentaje
desarrollo	100
humanidad	70
ingenieria	50
tecnología	40
sociedad	50

También se crearon 50 documentos artificialmente construidos para presentar un perfil que se encuentra entre el perfil del agente (mencionado arriba) y el siguiente, que representa las preferencias actuales del usuario :

Keyword	Porcentaje
mercado	60
comunicación	80
medios	30
publicidad	100
ventas	50

Los documentos que fueron traídos subsecuentemente por el sistema multiagentes fueron evaluados mediante el siguiente criterio (en sustitución del grado de interés de un usuario real) : se obtuvo manualmente el grado de proximidad entre el perfil de usuario y el documento encontrado, en forma de porcentaje (ver la sección *El sistema multiagentes : adaptación y aprendizaje*). Entonces se hizo coincidir ese porcentaje con una escala de calificación que va de -4 a 4, en donde -4 equivale a un grado de proximidad 0, en tanto que 4 equivale a un grado de proximidad 100. Esta escala de calificaciones corresponde a la escala que el usuario real puede asignar y que varía desde "nada interesante" hasta "interesantísimo".

Los resultados obtenidos se muestran en la siguiente tabla. La tabla muestra la cantidad de documentos obtenidos en cada excursión del agente Descubridor, así como el grado de proximidad, y la calificación asignada a cada documento.

Excursión	Documentos	Proximidad	Calificación
1	4	0, 6, 11	-4, -4, -3
2	3	16, 22, 26	-3, -2, -2
3	5	30, 34, 36, 38, 40	-2, -1, -1, -1
4	4	43, 46, 48, 50	-1, 0, 0, 0
5	3	51, 52, 55	0, 0, 1
6	3	56, 57, 60	1, 1, 1
7	4	63, 65, 66, 68	1, 1, 2, 2
8	4	70, 71, 76, 78	2, 2, 2, 3

9	3	80, 82, 84	3, 3, 3
10	6	88, 90, 92, 95, 97, 100	4, 4, 4, 4, 4, 4
11	0	-	-
12	0	-	-

De los resultados que la tabla muestra podemos remarcar los siguientes hechos :

- La representación interna del usuario que posee el agente, cambia efectivamente de acuerdo con la retroalimentación recibida.
- Dicha representación no cambia a menos que las preferencias del usuario cambien (como lo muestran las iteraciones 11 y 12).
- Los cambios en la representación de las preferencias del usuario ocurren abruptamente cuando los documentos sugeridos reciben calificaciones muy bajas.
- Los cambios en la representación de las preferencias del usuario ocurren lentamente cuando los documentos sugeridos reciben calificaciones altas.

Tiempo de ajuste de perfil en el sistema completo.

Como en el experimento anterior, se creó un perfil actual del usuario, definido por cinco conjuntos de palabras clave de cinco keywords cada uno, y se crearon artificialmente cinco agentes Intermediarios cuyo perfil de usuario era totalmente diferente.

Se crearon 50 documentos artificiales divididos en cinco conjuntos, colocados en cinco servidores. Cada conjunto de diez documentos contiene documentos cuyo contenido varía entre el perfil que conocen los agentes, y el perfil actual del usuario (que se supone fijo).

Los resultados obtenidos se muestran en la siguiente tabla. La tabla muestra la cantidad de documentos obtenidos en cada excursión de alguno de los agentes Descubridores, sin importar el tema al cual ese agente se dedica. También se muestra el grado de proximidad de cada documento, y la calificación asignada.

Excursión	Documentos	Proximidad	Calificación
1	3	0, 4, 16	-4, -4, -3
2	4	0, 5, 12	-4, -4, -3
3	4	0, 9, 18, 22	-4, -4, -3, -2
4	3	0, 10, 23	-4, -4, -2
5	4	12, 18, 27, 40	-4, -3, -2, -1
6	3	0, 10, 20	-4, -4, -3
7	3	22, 34, 53	-2, -1, 0
8	3	31, 41, 50	-2, -1, 0
9	4	21, 34, 43, 54	-2, -1, 0, 0
10	5	52, 64, 72, 83, 98	0, 1, 2, 3, 4
11	4	53, 62, 80, 89	0, 1, 3, 4
12	3	59, 71, 82, 95	1, 2, 3, 4
13	5	56, 67, 76, 90, 100	1, 2, 2, 3, 4
14	4	58, 68, 79, 92	1, 2, 3, 4

Estos resultados muestran una convergencia relativamente rápida. Aunque se requiere que el usuario lea y evalúe varias veces documentos de poco interés, finalmente el contenido de los documentos se ajusta a sus preferencias. A partir de ese momento, todos los documentos encontrados por el sistema tendrán un grado de interés no menor a "interesante"

A pesar de que la representación interna de las preferencias del usuario en cada agente Descubridor cambia cada vez que éste regresa al servidor hogar, en ocasiones este cambio no es suficiente para evitar que un documento sea seleccionado en dos incursiones consecutivas (hay que recordar que los agentes Descubridores no llevan un registro de los documentos que ya han encontrado y seleccionado anteriormente).

Parte V, Conclusiones

En este documento hemos presentado un sistema multiagentes cuya arquitectura y la arquitectura interna de cada uno de los integrantes permiten la realización de una tarea de naturaleza altamente distribuida en un dominio vasto. Hemos mostrado cómo se puede implementar una representación abstracta de las preferencias del usuario y cómo esa representación puede ser seccionada y distribuida entre un grupo homogéneo de agentes pertenecientes al sistema. Esta descomposición resulta en el uso de mecanismos de interacción local que permiten llegar a una solución global coherente. La aplicación de esta metodología ha permitido obtener un alto desempeño en la búsqueda distribuida de documentos html obedeciendo a las preferencias de una persona. Sin embargo, es la búsqueda asíncrona de información la que se ve mayormente beneficiada, es decir, aquellas situaciones en las que los documentos pueden ser presentados al usuario un poco mas adelante, por ejemplo, en el transcurso de una investigación. Para impactar positivamente la búsqueda de información recientemente disponible en línea, y que es requerida inmediatamente, será necesario incrementar la sofisticación del sistema multiagentes, sobre todo en cuanto a la variedad y el grado de especialización de los agentes del nivel mas bajo de la jerarquía.

Es evidente que la implementación actual de este sistema multiagentes proporciona resultados satisfactorios únicamente si se limita su radio de acción a una red de pocos nodos (las pruebas fueron

realizadas con un máximo de cinco servidores además del servidor hogar). Para poder aspirar a utilizar un sistema como el propuesto aquí, en una red tan amplia como internet, conservando la velocidad de ajuste a las preferencias del usuario, y sobre todo, la velocidad de respuesta a consultas explícitas, sería necesario construir sobre la arquitectura e implementación actuales, un conjunto adicional de estructuras de conocimiento, mecanismos de aprendizaje, y relaciones inter-agentes, que incrementaría la sofisticación del sistema.

Por otra parte, un modelo mas completo y exacto del usuario permitiría por una parte realizar con mas precisión la búsqueda de documentos en la red, y por otra, expandir la variedad de servicios que el sistema proporciona.

La facilidad y rapidez con la que el sistema multiagentes, y cada agente en particular, convergen hacia los tópicos de interés del usuario, aunada a la posibilidad de mantener una base de conocimientos automáticamente actualizada y siempre apegada a la realidad, permiten vislumbrar la gran importancia que el paradigma de agentes inteligentes aplicados a internet tendrá en un futuro próximo.

Apéndice A, Descripción de clases

Central.

Descripoón : Constituye el agente Central
Paquete : agentes
Extiende : Aglet.

Variables :

private VentanaCentral window : Ventana del agente que constituye la interfaz con el usuario.
private Vector agletProxies : El conjunto de proxies de los agentes Intermedianos
private Vector queryDocs : Documentos que responden a las consultas del usuario.
private DocumentPool documents : Documentos sugeridos al usuario
private AgentProperties prop . Propiedades del agente

Métodos :

public void onCreation (Object)
Se ejecuta en el momento de creación del agente.
Argumentos :
Objeto cualquiera para la inicalizaoón del agente

private final void loadProperties()
Carga las propiedades del agente del disco duro.

private void createAgents()
Crea los agentes Intermedianos con los que cuenta el sistema.

public void createNewAgent(Query)
Crea un nuevo agente Intermediano.
Argumentos :
Consulta sobre la cual trabajará el nuevo agente.

public synchronized void onDispatching(URL)
Se ejecuta cuando el agente va a ser despachado a otra localidad.
Argumentos :
URL al cual el agente se despachará

public void onReverting(URL)
Se ejecuta cuando el agente es retraido.
Argumentos :
URL donde el agente se encuentra.

public void onDeactivating(Date)
Se ejecuta cuando el agente es desactivado

public void onActivation(Date)
Se ejecuta cuando el agente es activado.

public synchronized void onDisposing()
Se ejecuta cuando el agente va a ser destruido

public void run()
Punto de entrada a la ejecución del agente.

protected synchronized void Dialog()
Muestra o trae al frente la ventana del agente.

public void callAgent(String)
Pide a un agente subordinado mostrar su ventana.
Argumentos :
nombre del agente subordinado.

public final void sendDocumentBack(String, int)
Envia un documento de vuelta al agente Intermediario correspondiente.
Argumentos :
nombre del documento
calificación asignada al documento

public final void solveQuery(Query)
Solicita a todos los agentes Intermedianos responder a una consulta.
Argumentos :
Consulta a responder

public final void showQueryDocs(String)
Muestra los documentos que responden a una consulta.
Argumentos :
Texto de la consulta a responder.

public boolean handleMessage(Message)
Manejador de mensajes de entrada.
Argumentos :
Mensaje recibido

protected synchronized void setTheMessage(String)
Pone la línea en la ventana como mensaje.
Argumentos :
Mensaje a desplegar.

Intermediario

Descripción : Agente Intermediario
Paquete : agentes
Extiende : Aglet

Variables :

VentanaIntermediario window : Ventana del agente
private AgentProperties prop : Propiedades y preferencias
private Vector IDs : Identificadores de los Descubridores
private DocumentPool documents : Documentos registrados
private newDocuments : Documentos nuevos (no evaluados)
private int sent : Número de documentos sugeridos

Métodos :

public void onCreation (Object)
Se ejecuta en el momento en que el agente es creado.
Argumentos :
Un objeto cualquiera para la inicialización del agente.

private void createDescubridor()
Crea un nuevo agente Descubridor

public final void callAgent(String)
Indica al agente Descubridor indicado que el usuario desea ver su ventana.
Argumentos :
Identificador del agente a llamar.

public synchronized void onDispatching(URL)
Se ejecuta cuando el agente va a ser despachado a otra localidad.
Argumentos :
Localidad destino.

public void onReverting(URL)
Se ejecuta cuando el agente es retraido
Argumentos :
Localidad de donde el agente es llamado.

public void onDeactivating(Date)
Se ejecuta cuando el agente es desactivado
Argumentos :
Tiempo de desactivación.

public void onActivation(Date)
Se ejecuta cuando el agente es activado

public synchronized void onDisposing()
Se ejecuta cuando el agente es destruido

public void run()
punto de entrada a la ejecución del agente.

public final void closeWindow()
Cierra la ventana del agente

public final void onWindowClosing()
Notifica al agente Descubridor que está cerrando su ventana.

private final void sendPreferences(AgletID)
Envia al agente Descubridor un perfil de usuario actualizado.
Argumentos :
  Identificador del agente Descubridor

private final void receiveNewDocuments(DocumentPool)
Recibe nuevos documentos del agente Descubridor y distribuye aquellos que no son útiles.
Argumentos :
  Documentos recibidos del agente Descubridor

private final void receiveRejectedDocuments(DocumentPool)
Recibe y revisa documentos rechazados por otros agentes
Argumentos :
  Documentos recibidos.

private final void askForEvaluation()
Pide al agente Central mostrar al usuario un documento.

private final void receiveEvaluatedDocument(Document)
Recibe un documento calificado del agente Central
Decide si registrarlo y actualiza el perfil de usuario
Actualiza el conjunto de documentos registrados de acuerdo
al nuevo perfil.
Argumentos :
  Documento evaluado recibido.

public final void solveQuery(Query)
Envia al agente Central documentos que pueden satisfacer la consulta
Argumentos :

Consulta a la que el agente responde.

Manejador de mensajes.
public boolean handleMessage(Message)
Argumentos :
Mensaje recibido.

Despliega la línea como mensaje
protected synchronized void setTheMessage(String)
Argumentos :
Mensaje a desplegar.

Descubridor

Descripción : Agente Descubridor
Paquete : agentes
Extiende : Aglet

Variables :

private VentanaDescubridor window;	ventana del agente
private DocumentPool documents;	conjunto de documentos encontrados
private Vector visitedSites;	conjunto de sitios ya visitados
private AgentProperties prop;	propiedades del agente
private boolean ack, arq;	variables de control
private AgletInfo info;	Información sobre el agente
private Status beliefs;	Estado del agente
private Goal mainGoal;	Meta principal (árbol de metas)

Métodos :

public void onCreation (Object)
Se ejecuta en el momento de creación del agente
Argumentos :
Objeto cualquiera para la inicialización del agente

Se ejecuta cuando el agente será despachado a otra localidad
public synchronized void onDispatching(MobilityEvent)
Argumentos :
Evento de mobilidad generado.

public void onReverting(MobilityEvent)
Se ejecuta cuando el agente será retraido
Argumentos :
Evento de mobilidad generado

public void onArrival(MobilityEvent)
Se ejecuta cuando el agente llega a un sitio
Argumentos :
  Evento de mobilidad generado

public void onDeactivating(Date)
Se ejecuta cuando el agente es desactivado
Argumentos :
  Tiempo de desactivación

public void onActivation(Date)
Se ejecuta cuando el agente es activado

public synchronized void onDisposing()
Se ejecuta cuando el agente es destruido

public final Status getEnvironmentStatus(Status)
Sensa las variables del ambiente especificadas en "st"
Argumentos :
  Un estado que contiene las variables de las que se desea obtener información.

public final void establishStatus()
Acutaliza la representación del estado del agente

private final void fillMainGoal()
Inicializa el árbol de metas

public void run()
Punto de entrada a la ejecución del agente

private final boolean atHome()
Indica si el agente se encuentra en el hogar (punto de creación)

private final String getNewSite()
Regresa una dirección nueva a donde el agente puede viajar

private final void reportMaster()
Envía documentos al agente Intermediario

private final void selectDocuments()
Selecciona documentos utilizando el perfil de usuario

private final void loadAnyDocument(int)
Selecciona documentos cualesquiera hasta tener por lo menos tres

Manejador de mensajes
public boolean handleMessage(Message)
Argumentos :
  Mensaje recibido.

Despliega la linea como mensaje.
protected synchronized void setTheMessage(String)
Argumentos :
  Cadena a desplegar como mensaje.

AgentProperties

Descripción : Esta clase constituye un conjunto de datos acerca de las propiedades del agente, asi como las preferencias del usuario que le corresponden. Esta clase permite guardar todos esos datos en disco duro, para leerlos de nuevo mas tarde.
Paquete : agentes.conocimiento
Extiende : java.util.Properties

Variables :

private AgletID masterID, Identificador del agente superior
private String fileOnDiskName: Nombre del archivo de propiedades

Métodos :

Constructor
public AgentProperties(AgletID, String)
Argumentos :
  Identificador del agente superior al propietario
  Nombre de archivo que le corresponde

public final AgletID getMasterID()
public final void setMasterID(AgletID ID) {
public final String getFilename() {

public final KeywordSet getUserKeywords()
Regresa el conjunto de keywords del usuario

public final void setUserKeywords(KeywordSet)
Establece el conjunto de keywords del usuario
Argumentos :
  Conjunto de keywords

public final int getNumAgents()

Regresa el número de agente con el que debe contar el agente

public final void setNumAgents(int)
Establece el número de agentes subordinados del agente
Argumentos :
  El número de agentes.

public void load()
Carga las propiedades desde el archivo en disco

public void save()
Guarda las propiedades en disco

public final void adjustToQuery(Query)
Establece el conjunto de keywords del usuario a partir de los keywords que forman una consulta.

DPlanLibrary

Descripción : Clase que constituye una tabla de decisión para un agente. Permite obtener planes de acción adecuados tanto al estado presente, que permitan alcanzar el estado deseado.
Paquete : agentes.conocimiento
Extiende : java.util.Vector.

Variables :

private Descubridor agent: Apuntador al agente Descubridor propietario

Métodos

Constructor :
public DPlanLibrary(Descubridor)
Argumentos :
  Agente propietario

private final void fill()
Inicializa la librería de planes

private final void addEntry(PlanLibraryEntry)
Añade a la librería una nueva entrada
Argumentos :
  La nueva entrada a la librería de planes

public final Plan getPlanFor(Status)
Indica el plan a seguir para lograr un estado deseado
Argumentos :

El estado deseado.

Goal

Descripción : Clase que constituye un árbol de metas y submetas. Consta de un nombre, un estado representativo, y submetas.
Paquete : agentes.conocimiento
Extiende : java.util.Vector

Variables :

```
private String name,       // nombre de la meta
 private Status status;     // Estado que la representa
 private boolean satisfied.  // ¿Ha sido lograda?
```

Métodos

Constructor :
public Goal(String. Status. booleaan)
Argumentos :
Nombre de la meta
Estado representativo
Bandera que indica si la meta ha sido lograda

public final void addSubGoal(Goal)
Añade una submeta
Argumentos :
Meta a añadir como submeta

public final String getName()
public final boolean isSatisfied()
public final void setSatisfied()
public final void setNotSatisfied()

public final Goal getImmediateGoal()
Obtiene la submeta que debe alcanzarse inmediatamente

public final Status getStatus()
Obtiene el status que representa a esta meta.

public final Status getDesiredStatus()
Obtiene el status deseado que representa a la meta inmediata.

public final void updateWith(Status)

Etiqueta cada meta como lograda si se ha logrado el status que esta indica, de lo contrario se marca como no lograda.
Argumentos :
El estado actual.

public final void reset()
Etiqueta todas las metas y submetas como no logradas.

Plan

Descripción : Esta clase constituye un conjunto de acciones a tomar. Es utilizada por la libreria de planes.
Paquete : agentes.conocimiento
Extiende : java.util.Vector

Metodos :

Constructor :
public Plan()

public void addAction(String)
Añade una acción al final del plan.
Argumentos :
El nombre de la acción

public final String actionAt(int)
Retorna la acción que ocupa el lugar indicado en el plan.

PlanLibraryEntry

Descripción : Esta clase constituye un conjunto de un conjunto de acciones a tomar para conseguir pasar de un estado actual a un estado deseado, dado un estado determinado del ambiente. Es utilizada por la libreria de planes.
Paquete : agentes.conocimiento

Variables :

private Status currentStatus; Estado actual del ambiente
private Status desiredStatus; Estado deseado del agente
private Plan actions; Conjunto de acciones a tomar

Métodos

Constructor :
public PlanLibraryEntry(Status, Status, Plan)

Argumentos :
El estado del ambiente al cual se aplica el plan
El estado que se desea lograr mediante este plan
El plan

public final boolean matches(Status, Status)
Indica si este plan es adecuado dados los estados actual y deseado.
Argumentos :
Estado actual
Estado deseado

public final boolean enables(Status)
Indica si este plan ayudaría a conseguir el estado indicado.
Argumentos :
Estado deseado.

public final Plan getPlan()
Retorna el plan.

public final Status getCurrentStatus()
Retorna el estado especificado como estado actual.

Status

Descripción : Clase que constituye una serie de atributos del ambiente y del agente y sus valores correspondientes.
Paquete : agentes.conocimiento
Extiende : java.util.Hashtable

Métodos :

Constructor :
public Status()

public final boolean contains(Status)
Indica si el estado especificado es un subconjunto de este estado

public final boolean isContainedIn(Status)
Indica si este estado es un subconjunto del estado especificado

public final boolean equals(Status)
Indica si el estado especificado es igual a este estado

public final Status add(Status)
Conjunta dos representaciones de estados diferentes. Los valores de este estado tienen prioridad.

Document

Descripción : Esta clase representa un documento HTML.
Paquete : agentes.documentos
Implementa : Serializable

Variables :

private KeywordSet keywords; keywords que representan al documento
private int grade; calificación asignada por el usuario
private AgletID ownerAgletID; Identificador del agente propietario
private String path; ruta del documento

Métodos

Constructor :
public Document(String)
Argumentos :
Ruta del documento.

public final void qualify(int)
Asigna la calificación especificada al documento.

Retorna la calificación del documento
public final int getGrade()

Establece la identidad del agente propietario del documento
public final void setOwnerAgletID(AgletID)
Argumentos :
Identificador del agente propietario

Retorna la identidad del agente propietario
public final AgletID getOwnerAgletID()

private final void extractKeywords() throws IOExceptión
Extrae las palabras clave del documento

public final KeywordSet getKeywords() throws IOExceptión
Retorna las palabras clave del documento.

public final int getProximity(KeywordSet) throws IOExceptión
Retorna el nivel de proximidad entre el documento y el conjunto de keywords especificado en una escala del 0 al 100.

public final String getPath()
Retorna la ruta del documento

public final void setURL(String url)
Establece el URL especificado como parte de la ruta del documento.

DocumentPool

Descripción : Esta clase constituye un conjunto de documentos html, y es utilizada por los agentes móviles e intermediarios.
Paquete : agentes.documentos
Extiende : java.util.Vector

Variables :

private Query query: Consulta a la que el conjunto de documentos responde.

Métodos :

Constructor :
public DocumentPool()

Añade un documento a la colección
public void addDocument(Document)

Elimina un documento de la colección
public boolean removeDocument(String)
Argumentos :
La ruta del documento a añadir.
Retorna true si el documento es eliminado

public final Document documentAt(int)
Retorna el documento que se encuentra en la posicion señalada.

public final int indexOf(String)
Retorna el índice del documento cuyo nombre se especifica.

public final Document getDocument(String)
Retorna el documento cuyo nombre se especifica.

public final DocumentPool addPool(DocumentPool)
Añade otro conjunto de documentos al final de éste.
Argumentos :
La colección de documentos a añadir.

public final void setOwnerAgletID(AgletID)
Establece la identidad del agente propietario de la colección.

public final AgletID getOwnerAgletID()
Retorna la identidad del agente propietario de la colección.

public final void setCausalQuery(Query)
Establece la consulta que originó la coloección de documentos.
Argumentos :
La consulta.

public final Query getCausalQuery()
Retorna la consulta que originó la coloección de documentos.

public final DocumentPool select(KeywordSet, int)
Selecciona los documentos que satisfagan el conjunto de keywords especificado con al menos un grado de proximidad como el especificado (en una escala del 0 al 100)
Argumentos :
Keywords que los documentos deben contener
Grado de proximidad requerido.

Keyword

Descripción : Esta clase constituye un keyword o palabra clave.
Paquete : agentes.documentos
Implementa : Serializable

Variables :

private String word, La palabra clave
private int rate; El porcentaje de ocurrencia dentro del documento en relación con las demás palabras
private int times; Número de ocurrencias dentro del documento

Métodos

Constructor :
public Keyword(String)
Argumentos :
Palabra clave

Constructor :
public Keyword(String, int) {
Argumentos :

Palabra clave
porcentaje de ocurrencia.

public final String getWord()
Retoma la palabra clave.

public final int getTimes()
Retoma el número de ocurrencias de la palabra.

public final void incTimes()
Incrementa la cuenta de ocurrencias de la palabra.

public final int getRate()
Retoma el porcentaje de ocurrencia de la palabra.

public final void setRate(int)
Establece el porcentaje de ocurrencia de la palabra.

public final void normalize(int max, int min) {
Establece el porcentaje de ocurrencia de la palabra a partir de el número de veces que ocurre en el documento y respecto a un mínimo y un máximo.

KeywordSet

Descripción : Esta clase constituye un conjunto de palabras clave en un documento html.
Paquete : agentes.documentos
Extiende : java.util.Vector
Implementa : Serializable

Métodos :

Constructor :
public KeywordSet()

Constructor :
public KeywordSet(String)
Construye el conjunto de keywords a partir de una cadena que los contiene

public boolean removeKeyword(Keyword)
Elimina un keyword de la colección.
Argumentos :
El keyword a eliminar.

public final Keyword keywordAt(int)
Retoma el keyword que se encuentra en la posición especificada de la colección.

public final int indexOf(String)
Retorna la posición de la palabra clave especificada.

public final Keyword getKeyword(String)
Retorna el keyword que contenga la palabra especificada

public final void addSet(KeywordSet, int)
Pondera los porcentajes de cada uno de los keywords del conjunto especificado con la calificación especificada, y entonces añade ese conjunto a este. Las palabras claves que se repitan en ambos conjuntos son conjuntadas en una sola y los porcentajes son sumados. Al final de la adición, el conjunto de keywords resultante es normalizado.

public final void normalize()
Normaliza los porcentajes de cada uno de los keywords en relación al porcentaje mayor del conjunto

public final String toText()
Retorna una representación en cadena del conjunto de keywords.

public final int getProximity(KeywordSet ks)
Retorna una medida (en la escala del 0 al 100) en que el conjunto de palabras clave especificado se asemeja en contenido y distribución a éste conjunto.

Query

Descripción : Esta clase constituye un conjunto de parámetros que definen una consulta del usuario. Nota. en esta versión el único parámetro que define a una consulta es un conjunto de keywords.
Paquete : agentes.documentos
Extiende : KeywordSet

Variables :

private Date timeStamp: Momento de elaboración de la consulta por parte del usuario.

Métodos :

Constructor :
public Query(String qs)
Argumentos :
Cadena que contiene el conjunto de palabras clave.

Apéndice B, Código Fuente

Con el objeto de brindar al lector un mejor entendimiento del funcionamiento del sistema, así como de facilitar la implementación futura de aplicaciones similares, se incluye a continuación el código fuente escrito en Java que conforma el sistema multiagentes.

Por razones de espacio se dejaron fuera de este compendio aquellas clases que no resultan cruciales para la comprensión de los programas, y que son bastante fáciles de implementar.

Agente Central

```
/* @(#)Central.java */

package agentes;

import aglet.*;
import ibm.aglets.util.*;
import agentes.ventanas.*;
import agentes.conocimiento.*;
import agentes.documentos.*;
import java.net.*;
import java.io.*;
import java.util.*;

/**
 * Esta clase hereda de Aglet. Constituye el comportamiento
 * del agente Central.
 *
 * @version      1.00    Julio/1997
 * @author       Elohim Puon
 */

public class Central extends Aglet {
    VentanaCentral window;              // La ventana del agente
    private Vector agletProxies;        // El conjunto de proxies de los
agentes Intermediarios
    private Vector queryDocs;           // Documentos que responden a una
consulta
    private DocumentPool documents;     // Documentos sugeridos al usuario
    private AgentProperties prop;       // Propiedades del agente

    //
    // Se ejecuta en el momento de creación del agente
    //
    public void onCreation (Object o) {
        try {
            window = new VentanaCentral(this);
        } catch (Exception e) {
            inError(e.getMessage());
```

```
        }
        loadProperties();
        agletProxies = new Vector();
        createAgents();
        documents = new DocumentPool();
        queryDocs = new DocumentPool();
    }

    //
    // Carga las propiedades del agente del disco duro
    //
    private final void loadProperties() {
      prop = new AgentProperties(getIdentifier(), "agents");
      prop.load();
    }

    //
    // Crea los agentes Intermediarios con los que cuenta el sistema
    //
    private void createAgents() {
      NameTable names = new NameTable();
      setTheMessage("Voy a crear los agentes Intermediarios");
      try {
        for(int i=0; i<prop.getNumAgents(); i++) {
          AgentProperties aProp = new AgentProperties(getIdentifier(),
names.getHumanName(i));

agletProxies.addElement(getAgletContext().createAglet(getCodeBase(),
                                    "agentes.Intermediario", aProp));
          window.addAgent(aProp.getFilename());
        }
        setTheMessage("Agentes Intermediarios creados");
      } catch(Exception e) {
        setTheMessage(e.getMessage());
        prop.setNumAgents(agletProxies.size());
        setTheMessage("Sólo se pudieron crear " + Integer.toString
                      (prop.getNumAgents()) + " agentes");
      }
    }

    //
    // Crea un nuevo agente Intermediario
    //
    public void createNewAgent(Query q) {
      NameTable names = new NameTable();
      setTheMessage("Voy a crear un nuevo agente Intermediario");
      try {
        AgentProperties aProp = new AgentProperties(getIdentifier(),

names.getHumanName(agletProxies.size()));
        aProp.adjustToQuery(q);
```

```
        aProp.save();

agletProxies.addElement(getAgletContext().createAglet(getCodeBase(),
                                "agentes.Intermediario", aProp));
        window.addAgent(aProp.getFilename());
        setTheMessage("Nuevo agente: " + aProp.getFilename());
      } catch(Exception e) {
        setTheMessage("No se pudo crear el nuevo agente");
        e.printStackTrace();
      }
    }

    //
    // Se ejecuta cuando el agente va a ser despachado a otra localidad
    //
    public synchronized void onDispatching(URL URL) {
       throw new SecurityException("¡Soy un agente estacionario!");
//Aglet inmóvil
    }

    //
    // Se ejecuta cuando el agente es retraido
    //
    public void onReverting(URL remoteURL) {
      throw new SecurityException();
    }

    //
    // Se ejecuta cuando el agente es desactivado
    //
    public void onDeactivating(Date date) {
      window.show(false);
    }

    //
    // Se ejecuta cuando el agente es activado
    //
    public void onActivation(Date date) {
      window.show(true);
    }

    //
    // Se ejecuta cuando el agente va a ser destruido
    //
    public synchronized void onDisposing() {
      if (window != null) window.dispose();
      Message m = new Message("terminating");
      try {
        getAgletContext().multicastMessage(m);
      } catch(Exception ex) {
        setTheMessage(ex.getMessage());
```

```
            ex.printStackTrace();
        }
        prop.setNumAgents(agletProxies.size());
        prop.save();
    }

    //
    // Punto de entrada a la ejecución del agente
    //
    public void run() {
        setText("Iniciando");
        try { getMessageManager().exitMonitor(); } catch (Exception e)
{}
    }

    //
    // Muestra o trae al frente la ventana del agente
    //
    protected synchronized void Dialog() {
        if (window.isShowing) window.show(false);
        else window.show(true);
    }

    //
    // Pide a un agente subordinado mostrar su ventana
    //
    public void callAgent(String name) {
        NameTable names = new NameTable();
        Message m = new Message("showWindow");
        try {

((AgletProxy)agletProxies.elementAt(names.getAgentNumber(name))).sendM
essage(m);
        } catch(Exception e) {
            setTheMessage(e.getMessage() + " " + e.toString());
            e.printStackTrace();
        }
    }

    //
    // Envia un documento de vuelta al agente Intermediario
correspondiente.
    //
    public final void sendDocumentBack(String docPath, int grade) {
        Document d = documents.getDocument(docPath);
        if(d != null) {
            d.qualify(grade);
            Message m = new Message("evaluatedDocument", d);
            try {

getAgletContext().getAgletProxy(d.getOwnerAgletID()).sendMessage(m);
```

```
            documents.removeDocument(d.getPath());
            window.removeDocument(docPath);
          } catch(Exception e) {
            setTheMessage(e.getMessage());
            e.printStackTrace();
          }
        }
      }

      //
      // Solicita a todos los agentes Intermediarios responder a una
consulta
      //
      public final void solveQuery(Query query) {
        Message m = new Message("solveQuery", query);
        try {
          getAgletContext().multicastMessage(m);
        } catch(Exception e) {
          setTheMessage(e.getMessage());
          e.printStackTrace();
        }
      }

      //
      // Muestra los documentos que responden a una consulta
      //
      public final void showQueryDocs(String queryText) {
        DocumentPool docToShow = new DocumentPool();
        docToShow.setCausalQuery(new Query(queryText));
        for(int i=0; i < queryDocs.size(); i++) {
          DocumentPool docs = (DocumentPool)queryDocs.elementAt(i);
          if (docs.getCausalQuery().toText().equals(queryText)) {
            docToShow.addPool(docs);
            queryDocs.removeElementAt(i);
          }
        }
        setTheMessage(Integer.toString(docToShow.size()) + " documentos
satisfacen la consulta.");
        QueryResultWindow w = new QueryResultWindow(this, docToShow);
      }

      //
      // Manejador de mensajes de entrada
      //
      public boolean handleMessage(Message msg) {  //manejador de
mensajes
        try {
          if(msg.sameKind("error")) inError((String)(msg.getArg()));
          else if(msg.sameKind("evaluateDocuments")) {
            DocumentPool nd = (DocumentPool)msg.getArg();
            documents.addPool(nd);
```

```
            for(int i=0; i<nd.size(); i++) {
              window.addDocument(nd.documentAt(i).getPath());
            }
            setTheMessage("Se recibieron nuevos documentos");
            return true;
          }
          else if(msg.sameKind("proposedDocuments")) {
            DocumentPool nd = (DocumentPool)msg.getArg();
            boolean exists = false;
            for(int i=0; i < queryDocs.size(); i++) {
              DocumentPool dp =(DocumentPool)queryDocs.elementAt(i);
              if
(dp.getCausalQuery().toText().equals(nd.getCausalQuery().toText())) {
                exists = true;
                dp.addPool(nd);
              }
            }
            if(!exists) queryDocs.addElement(nd);
            setTheMessage("Recibí respuesta a la consulta: " +
nd.getCausalQuery().toText());
            return true;
          }
          else return false;
          return false;
        } catch (Exception e) {
          // -- not yet handled
          setTheMessage(e.getMessage());
          e.printStackTrace();
        }
        return false;
      }

      protected synchronized void setTheMessage(String text) { // pone la
línea en la ventana
        if (window != null)
          window.setMessage(text);
      }

      protected synchronized void inError(Object message) {
setTheMessage((String)message); }

}
```

Agente Intermediario

```
/* @(#)Intermediario.java */

package agentes;

import aglet.*;
```

```
import ibm.aglets.util.*;
import agentes.ventanas.*;
import agentes.conocimiento.*;
import agentes.documentos.*;
import java.net.*;
import java.io.IOException;
import java.util.Vector;
import java.util.Properties;
import java.util.Enumeration;
import java.util.Date;


/**
 * Esta clase hereda de Aglet. Constituye el comportamiento
 * los agentes intermediarios.
 *
 * @see WriterSlave
 * @version      0.00    Julio/1997
 * @author       Elohim Puon
 */

public class Intermediario extends Aglet {
   VentanaIntermediario window;                          // Ventana del agente
   private AgentProperties prop;                         // Propiedades y
preferencias
   private Vector IDs;                                   // Identificadores de
los Descubridores
   private DocumentPool documents, newDocuments; // Documentos
registrados y nuevos
   private int sent;                                     // Número de
documentos sugeridos

   //
   // Se ejecuta en el momento en que el agente es creado.
   //
   public void onCreation (Object o) {
      prop = (AgentProperties)o;
      prop.load();
      IDs = new Vector();
      documents = new DocumentPool();
      newDocuments = new DocumentPool();
      sent = 0;
      try {
       subscribeMessage("updateWindow");
          subscribeMessage(prop.getFilename()+".newDocuments");
          subscribeMessage("solveQuery");
          subscribeMessage("terminating");
          subscribeMessage("rejectedDocuments");
      } catch (Exception e) {
      setTheMessage(e.getMessage());
      }
```

```
        try {
           window = new VentanaIntermediario(this, prop.getFilename());
        } catch (Exception e) {
           inError(e.getMessage());
        }
        createDescubridor();
    }

    //
    // Crea un nuevo agente Descubridor
    //
    private void createDescubridor() {
      setTheMessage("Keywords: " + prop.getProperty("keywords"));
      setTheMessage("rates: " + prop.getProperty("rates"));
      setTheMessage("Voy a crear el agente descubridor");
       try {
         AgletID newID =
getAgletContext().createAglet(getCodeBase(),"agentes.Descubridor",
prop).getAgletID();
         IDs.addElement(newID);
         window.addAgent(newID.toString());
         setTheMessage("Un nuevo agente descubridor ha sido creado");
       } catch(Exception e) {
         setTheMessage(e.getMessage());
       }
    }

    //
    // Indica al agente Descubridor indicado que el usuario desea ver
su ventana.
    //
    public final void callAgent(String ID) {
      try {
        setTheMessage("Llamando a " + ID);
        Message call = new Message("showWindow");
        for(int i=0; i<IDs.size(); i++)
          if(((AgletID)IDs.elementAt(i)).toString().equals(ID)) {

getAgletContext().getAgletProxy((AgletID)IDs.elementAt(i)).sendMessage
(call);
        }
      } catch(Exception e) {
        e.printStackTrace();
        setTheMessage("El agente se encuentra de viaje");
      }
    }

    //
    // Se ejecuta cuando el agente va a ser despachado a otra
localidad.
    //
```

```
   public synchronized void onDispatching(URL URL) {
      throw new SecurityException(";Soy un agente estacionario!");
//Aglet inmóvil
   }


   //
   // Se ejecuta cuando el agente es retraido
   //
   public void onReverting(URL remoteURL) {
     throw new SecurityException();
   }

   //
   // Se ejecuta cuando el agente es desactivado
   //
   public void onDeactivating(Date date) { window.show(false); }

   //
   // Se ejecuta cuando el agente es activado
   //
   public void onActivation(Date date) {
     window.show(true);
   }

   //
   // Se ejecuta cuando el agente es destruido
   //
   public synchronized void onDisposing() {
     prop.save();
     setTheMessage("Propiedades salvadas");
     closeWindow();
   }

   public void run() {    // punto de entrada a la ejecución del
agente.
      setText("Comenzando");
      try { getMessageManager().exitMonitor(); } catch (Exception e)
{}
   }

   protected synchronized void dialog() { //muestra o trae al frente
la ventana
     if (window.isShowing()) window.show(false);
     else window.show(true);
   }

   public final void closeWindow() {
     window.show(false);
     onWindowClosing();
   }
```

```
    //
    // Notifica al agente Descubridor que está cerrando su ventana.
    //
    public final void onWindowClosing() {
      setTheMessage("Notificando el cierre de ventana");
      Message msg = new Message("closingMasterWindow");
      try {
        for(int i=0; i<IDs.size(); i++)

getAgletContext().getAgletProxy((AgletID)IDs.elementAt(i)).sendMessage
(msg);
      } catch(Exception e) {
        setTheMessage(e.getMessage());
        e.printStackTrace();
      }
    }

    //
    // Envia al agente Descubridor un perfil de usuario actualizado.
    //
    private final void sendPreferences(AgletID id) {
      setTheMessage("Enviando nuevas preferencias");
      Message ack = new Message("newProfile", prop);
      try {
        getAgletContext().getAgletProxy(id).sendMessage(ack);
      } catch(Exception e) {
        setTheMessage(e.getMessage());
        e.printStackTrace();
      }
    }

    //
    // Recibe nuevos documentos del agente Descubridor
    // Distribuye aquellos que no son utiles.
    //
    private final void receiveNewDocuments(DocumentPool nd) {
      DocumentPool rejectedDocs = new DocumentPool();
      nd.setOwnerAgletID(getAgletID());
      try {
        for(int i=0; i<nd.size(); i++) {
          Document d = nd.documentAt(i);
          if(d.getProximity(prop.getUserKeywords()) > 40)
            newDocuments.addDocument(d);
          else rejectedDocs.addDocument(d);
        }
      } catch(IOException e) {
        setTheMessage(e.getMessage());
        e.printStackTrace();
      }
      if (rejectedDocs.size() > 0) {
```

```
        setTheMessage("Ofreciendo documentos a otros agentes");
        Message msg = new Message("rejectedDocuments", rejectedDocs);
        try {
          getAgletContext().multicastMessage(msg);
        } catch(Exception e) {
          setTheMessage(e.getMessage());
          e.printStackTrace();
        }
      }
    }

    private final void receiveRejectedDocuments(DocumentPool rd) {
      rd.setOwnerAgletID(getAgletID());
      newDocuments.addPool(rd.select(prop.getUserKeywords(), 40));
    }

    //
    // Pide al agente Central mostrar al usuario un documento.
    //
    private final void askForEvaluation() {
      DocumentPool docsToEvaluate = new DocumentPool();
      for(int i=0; (i<newDocuments.size()) && (sent<6); i++, sent++) {
        docsToEvaluate.addDocument(newDocuments.documentAt(i));
        newDocuments.removeElementAt(i);
      }
      setTheMessage("Documentos a evaluar: " + docsToEvaluate.size());
      if (docsToEvaluate.size() > 0) {
        setTheMessage("Enviando documentos para su evaluación: " +
docsToEvaluate.size());
        Message msg = new Message("evaluateDocuments", docsToEvaluate);
        try {

getAgletContext().getAgletProxy(prop.getMasterID()).sendMessage(msg);
        } catch(Exception e) {
          setTheMessage(e.getMessage());
          e.printStackTrace();
        }
      }
    }

    //
    // Recibe un documento calificado del agente Central.
    // Decide si registrarlo y actualiza el perfil de usuario.
    // Actualiza el conjunto de documentos registrados de acuerdo
    // al nuevo perfil.
    //
    private final void receiveEvaluatedDocument(Document d) {
      setTheMessage("Recibi un documento calificado: " + d.getPath());
      setTheMessage("Con calificación: " +
 Integer.toString(d.getGrade()));
      try {
```

```
        KeywordSet kw = prop.getUserKeywords();
        kw.addSet(d.getKeywords(), d.getGrade());
        prop.setUserKeywords(kw);
        if (d.getGrade() > 5) {
          documents.removeDocument(d.getPath());
          documents.addDocument(d);
          window.addDocument(d.getPath());
        }
      } catch(IOException e) {
        setTheMessage("No es posible determinar los keywords del
documento");
      }
      documents = documents.select(prop.getUserKeywords(), 30);
      if(sent > 0) sent--;
    }

    //
    // Envia al agente Central documentos que pueden satisfacer la
consulta
    //
    public final void solveQuery(Query query) {
      setTheMessage("Enviando documentos propuestos.");
      Message msg = new Message("proposedDocuments",
documents.select(query, 40));
      try {

getAgletContext().getAgletProxy(prop.getMasterID()).sendMessage(msg);
      } catch(Exception e) {
        setTheMessage(e.getMessage());
        e.printStackTrace();
      }
    }

    public boolean handleMessage(Message msg) {   //manejador de
mensajes
      setTheMessage("Mensaje recibido: " + msg.kind);
      try {
        if(msg.sameKind("error")) inError((String)(msg.getArg()));
        else if(msg.sameKind(prop.getFilename() + ".newDocuments")) {
          DocumentPool d = (DocumentPool)msg.getArg();
          AgletID owner = d.getOwnerAgletID();
          setTheMessage("Recibiendo " + d.size() + " documentos nuevos
de " + owner.toString());
          receiveNewDocuments(d);
          askForEvaluation();
          if(owner != null) sendPreferences(owner);
          return true;
        }
        else if(msg.sameKind("evaluatedDocument")) {
          receiveEvaluatedDocument((Document)msg.getArg());
          askForEvaluation();
```

```
            return true;
          }
          else if(msg.sameKind("rejectedDocuments")) {
            receiveRejectedDocuments((DocumentPool)msg.getArg());
            askForEvaluation();
            return true;
          }
          else if(msg.sameKind("solveQuery")) {
            solveQuery((Query)msg.getArg());
            return true;
          }
          else if(msg.sameKind("showWindow")) {
            window.show(true);
            return true;
          }
          else if(msg.sameKind("terminating")) {
            dispose();
            return true;
          }
          else return false;
          return false;
        } catch (Exception e) {
          // -- not yet handled
          setTheMessage("Exception at Intermediary agent:
"+e.getMessage());
          e.printStackTrace();
        }
        return false;
    }

    protected synchronized void setTheMessage(String text) { // pone la
linea en la ventana
        if (window != null)
          window.setMessage(text);
    }

    protected synchronized void inError(Object message) {
setTheMessage((String)message); }

}
```

Agente Descubridor

```
/* @(#)Descubridor.java */

package agentes;

import aglet.*;
import aglet.event.*;
import ibm.aglets.util.*;
```

```
import agentes.ventanas.*;
import agentes.documentos.*;
import agentes.conocimiento.*;
import java.net.*;
import java.io.*;
import java.util.Vector;
import java.util.Properties;
import java.util.Enumeration;
import java.util.Date;

/**
 * Esta clase hereda de Aglet. Constituye el comportamiento
 * los agentes descubridores de información.
 *
 * @version      1.00    Junio/1997
 * @author       Elohim Duon
 */

public class Descubridor extends Aglet implements MobilityListener {

  VentanaDescubridor window;          // ventana del agente
  private DocumentPool documents;     // conjunto de documentos
encontrados
  private Vector visitedSites;        // conjunto de sitios ya visitados
  private AgentProperties prop;       // propiedades
  private boolean ack, arq;           // variables de control
  private AgletInfo info;             // Información sobre el agente
  private Status beliefs;             // Estado del agente
  private Goal mainGoal;              // Meta principal (árbol de metas)

  //
  // Se ejecuta en el momento de creación del agente
  //
  public void onCreation (Object o) {
    documents = new DocumentPool();
    prop = (AgentProperties)o;
    window = new VentanaDescubridor(this, prop.getFilename());
    documents = new DocumentPool();
    prop = (AgentProperties)o;
    info = getAgletInfo();
    visitedSites = new Vector();
    visitedSites.addElement(info.getOrigin());
    addMobilityListener(this);
    beliefs = new Status();
    arq = false;
    ack = false;
    fillMainGoal();
  }

  //
  // Se ejecuta cuando el agente será despachado a otra localidad
```

```
//
public synchronized void onDispatching(MobilityEvent ev) {
  setTheMessage("Voy a " + ev.getLocation());
  window.show(false);
}

//
// Se ejecuta cuando el agente será retraido
//
public void onReverting(MobilityEvent ev) {
  throw new SecurityException("No permitido");
}

//
// Se ejecuta cuando el agente llega a un sitio
//
public void onArrival(MobilityEvent ev) {
  window.show(true);
}

//
// Se ejecuta cuando el agente es desactivado
//
public void onDeactivating(Date date) {
  window.show(false);
}

//
// Se ejecuta cuando el agente es activado
//
public void onActivation(Date date) {
  window.show(true);
}

//
// Se ejecuta cuando el agente es destruido
//
public synchronized void onDisposing() {
  window.dispose();
}

//
// Sensa las variables del ambiente especificadas en "st"
//
public final Status getEnvironmentStatus(Status st) {
  Status es = new Status();
  for(Enumeration e=st.keys(); e.hasMoreElements();) {
    String name = (String)e.nextElement();
    if(name.equals("Nuevo sitio disponible"))
      es.put(name, new Boolean(getNewSite() != null));
  }
```

```
      return es;
    }

    //
    // Acutaliza la representación del estado del agente
    //
    public final void establishStatus() {
      beliefs.put("He enviado nuevos documentos", new Boolean(arq));
      beliefs.put("He sido reconocido", new Boolean(ack));
      beliefs.put("Estoy en el hogar", new Boolean(atHome()));
      beliefs.put("Llevo suficientes documentos", new
 Boolean(documents.size()>2));
      beliefs.put("Estoy en un nuevo sitio", new Boolean(!visitedSites.

 contains(getAgletContext().getHostingURL().toString())));
    }

    //
    // Inicializa el árbol de metas
    //
    private final void fillMainGoal() {
      {
      Status st = new Status();
      st.put("He sido reconocido", new Boolean(true));
      mainGoal = new Goal("Obtener reconocimiento", st, false);
      }
        {
        Status st = new Status();
        st.put("Llevo suficientes documentos", new Boolean(true));
        mainGoal.getImmediateGoal().addSubGoal(new Goal("Llevar
sufucientes documentos", st, false));
        }
          {
          Status st = new Status();
          st.put("Estoy en un nuevo sitio", new Boolean(true));
          mainGoal.getImmediateGoal().addSubGoal(new Goal("Estar en un
nuevo sitio", st, true));
          }
        mainGoal.getImmediateGoal().setSatisfied();
        {
        Status st = new Status();
        st.put("Estoy en el hogar", new Boolean(true));
        mainGoal.getImmediateGoal().addSubGoal(new Goal("Estar en el
hogar", st, true));
        }
        {
        Status st = new Status();
        st.put("He enviado nuevos documentos", new Boolean(true));
        mainGoal.getImmediateGoal().addSubGoal(new Goal("", st, true));
        }
      mainGoal.setSatisfied();
```

```
      mainGoal.reset();
    }

    //
    // Punto de entrada a la ejecución del agente
    //
    public void run() {
      DPlanLibrary pl = new DPlanLibrary(this);
      setText("Iniciando");
      while (true) {
        try {getMessageManager().exitMonitor();} catch (Exception e) {}

        establishStatus();
        mainGoal.updateWith(beliefs);
        if (mainGoal.isSatisfied()) {
          arq = false;
          ack = false;
          establishStatus();
          mainGoal.reset();
          mainGoal.updateWith(beliefs);
        }
        if (mainGoal.getDesiredStatus() != null) {
          Plan p = pl.getPlanFor(mainGoal.getDesiredStatus());
          setTheMessage("Estado deseado: " +
mainGoal.getDesiredStatus().toString());
          for(int i=0; i<p.size(); i++) {
            setTheMessage(">>> acción: " + p.actionAt(i));
            if(p.actionAt(i).equals("Viaja a un nuevo sitio")) {
              try {
                dispatch(new URL(getNewSite()));
              } catch(Exception e) {
                setTheMessage(e.getMessage());
              }
            }
            else if(p.actionAt(i).equals("Selecciona documentos"))
              selectDocuments();
            else if(p.actionAt(i).equals("Lleva documentos
cualesquiera"))
              loadAnyDocument(3-documents.size());
            else if(p.actionAt(i).equals("Envia nuevos documentos"))
              reportMaster();
            else if(p.actionAt(i).equals("Regresa al hogar")) {
              try {
                dispatch(new URL(info.getOrigin()));
              } catch (Exception e) {
                setTheMessage(e.getMessage());
              }
            }
          }
        }
      }
```

```
    }

    //
    // Indica si el agente se encuentra en el hogar (punto de creación)
    //
    private final boolean atHome() {
return(info.getOrigin().equals(getAgletContext().getHostingURL().toStr
ing()));
    }

    //
    // Regresa una dirección nueva a donde el agente puede viajar
    //
    private final String getNewSite() {
      DataInputStream stream;
      Vector notVisitedSites = new Vector();
      String s = "";
      int missing;
      try {
        stream = new DataInputStream(new
FileInputStream("agents.addresses"));
        while((s = stream.readLine()) != null)
          if(!visitedSites.contains(s)) notVisitedSites.addElement(s);
        if((missing = notVisitedSites.size()) > 0)
          s =
(String)notVisitedSites.elementAt((int) (Math.random()*(missing-1)));
        stream.close();
      } catch (Exception e) {
        setTheMessage(e.getMessage());
      }
      return s;
    }

    //
    // Envia documentos al agente Intermediario
    //
    private final void reportMaster() {
      documents.setOwnerAgletID(getAgletID());
      Message mensaje = new Message(prop.getFilename()+".newDocuments",
documents);
      setTheMessage("Voy a entregar nuevos documentos..." +
documents.size());
      try {
        getAgletContext().multicastMessage(mensaje);
        setTheMessage("Entregué nuevos documentos. " +
documents.size());
        arg = true;
      } catch(Exception e) {
        setTheMessage(e.getMessage() + " " + e.toString());
        e.printStackTrace();
```

```
    }
  }

  //
  // Selecciona documentos utilizando el perfil de usuario
  //
  private final void selectDocuments() {
    setText("selecting");
    String path = "servidor" +
Integer.toString(getAgletContext().getHostingURL().getPort());
    File a = new File(path);
    String[] dir = a.list();
    setTheMessage("Obteniendo las concordancias");
    setTheMessage(dir.length + " archivos *.*");
    for(int doc=0; doc < dir.length; doc++)
      if(dir[doc].toLowerCase().indexOf(".htm") > -1) {
        Document d = new Document(path + File.separator + dir[doc]);
        if(documents.indexOf(d.getPath()) < 0) try {
          setTheMessage("Leyendo el documento: " + d.getPath());
          if(d.getProximity(prop.getUserKeywords()) > 40) {
            arq = false;
            d.setURL(getAgletContext().getHostingURL().toString());
            documents.addDocument(d);
            window.addDocument(d.getPath());
          }
        } catch(IOException e) {
          setTheMessage("No se puede establecer la proximidad de: " +
d.getPath());
        }
      }

if(!visitedSites.contains(getAgletContext().getHostingURL().toString()
))

visitedSites.addElement(getAgletContext().getHostingURL().toString());
    }

    //
    // Selecciona documentos cualesquiera hasta tener por lo menos tres
    //
    private final void loadAnyDocument(int numDocs) {
      setText("reading");
      String path = "servidor" +
 Integer.toString(getAgletContext().getHostingURL().getPort());
      File a = new File(path);
      String[] dir = a.list();
      for(int doc=0; (doc < dir.length) && (documents.size() < 3);
doc++)
        if(dir[doc].toLowerCase().indexOf(".htm") > -1) {
          Document d = new Document(path + File.separator + dir[doc]);
          arq = false;
```

```
            documents.addDocument(d);
            window.addDocument(d.getPath());
        }
    }

    public boolean handleMessage(Message msg) {   //manejador de
mensajes
      try {
        if("newProfile".equals(msg.kind)) {
          setTheMessage("Documentos enviados exitosamente.");
          documents.removeAllElements();
          window.removeAllDocuments();
          setTheMessage("Se recibieron nuevas preferencias");
          prop = (AgentProperties)msg.getArg();
          visitedSites.removeAllElements();
          visitedSites.addElement(info.getOrigin());
          ack = true;
          return true;
        }
        else if("showWindow".equals(msg.kind)) {
          setTheMessage("¿Que tal!");
          window.show(true);
          return true;
        }
        else if("closingMasterWindow".equals(msg.kind)) {
          window.show(false);
          return true;
        }
        else return false;
      } catch (Exception e) {
        // -- not yet handled
      }
      return false;
    }

    protected synchronized void setTheMessage(String text) { // pone la
linea en la ventana
      //super.setText(text);
      if (window != null)
        window.setMessage(text);
    }

    protected synchronized void inError(Object message) {
setTheMessage((String)message); }

}
```

Status

```
/* @(#)Status.java */
```

```
package agentes.conocimiento;

import java.util.Hashtable;
import java.util.Enumeration;

/**
 * Clase que constituye una serie de atributos y
 * sus valores correspondientes.
 *
 * @version     1.00    Julio/1997
 * @author      Elohim Puon
 */

public class Status extends Hashtable {

  public Status() {     // Constructor
    super();
  }

  //
  // Indica si el estado st es un subconjunto de este estado
  //
  public final boolean contains(Status st) {
    boolean c = true;
    for(Enumeration k = st.keys(); k.hasMoreElements();) {
      Object key = k.nextElement();
      c = (c && containsKey(key) && (get(key).equals(st.get(key))));
    }
    return c;
  }

  //
  // Indica si este estado es un subconjunto del estado st.
  //
  public final boolean isContainedIn(Status st) {
    boolean c = true;
    for(Enumeration k = keys(); k.hasMoreElements();) {
      Object key = k.nextElement();
      c = (c && st.containsKey(key) &&
(st.get(key).equals(get(key))));
    }
    return c;
  }

  //
  // Indica si el estado st es igual a este estado
  public final boolean equals(Status st) {
    return(contains(st) && isContainedIn(st));
  }
```

```
  //
  // Conjunta dos representaciones de estados diferentes.
  // Los valores de este estado tienen prioridad.
  //
  public final Status add(Status st) {
     for(Enumeration e = st.keys(); e.hasMoreElements();) {
       remove(e.nextElement());      // elimina redundancias
     }
     for(Enumeration e = st.keys(); e.hasMoreElements();) {
       Object name = e.nextElement();
       put(name, st.get(name));      // suma
     }
     return this;
  }

}
```

Goal

```
/* @(#)Goal.java */

package agentes.conocimiento;

import java.util.Vector;

/**
 * Clase que constituye un árbol de metas y submetas.
 * Consta de un nombre, un estado representativo, y submetas.
 *
 * @version      0.00    Julio/1997
 * @author       Elohim Pinon
 */

public class Goal extends Vector {
  private String name;          // nombre de la meta
  private Status status;        // Estado que la representa
  private boolean satisfied;    // ¿Ha sido lograda?

  public Goal(String nm, Status st, boolean sat) {     // Constructor
     super();
     name = nm;
     status = st;
     satisfied = sat;
  }

  public final void addSubGoal(Goal g) {  // añade una submeta
     super.addElement(g);
  }

  public final String getName() { return name; }
```

```
public final boolean isSatisfied() { return satisfied; }

public final void setSatisfied() { satisfied = true; }

public final void setNotSatisfied() { satisfied = false; }

// Obtiene la submeta que debe alcanzarse inmediatamente
public final Goal getImmediateGoal() {
  for(int i=0; i<size(); i++)
    if(!((Goal)elementAt(i)).isSatisfied())
      return ((Goal)elementAt(i)).getImmediateGoal();
  if(!this.isSatisfied()) return this;
  else return null;
}

// Obtiene el status que representa a esta meta.
public final Status getStatus() {
  return status;
}

// Obtiene el status deseado que representa a la meta inmediata.
public final Status getDesiredStatus() {
  Goal g = getImmediateGoal();
  if(g != null) return g.getStatus();
  else return null;
}

// Etiqueta cada meta como lograda si se ha
// logrado el status que esta indica, de lo contrario se
// marca como no lograda.
public final void updateWith(Status st) {
  if(getStatus().isContainedIn(st)) setSatisfied();
  else setNotSatisfied();
  for(int i=0; i<size(); i++) ((Goal)elementAt(i)).updateWith(st);
}

// Etiqueta todas las metas y submetas como no logradas.
public final void reset() {
  setNotSatisfied();
  for(int i=0; i<size(); i++) ((Goal)elementAt(i)).reset();
}

}
```

Plan

```
/* @(#)Plan.java */

package agentes.conocimiento;
```

```
import java.util.Vector;

/**
 * Esta clase constituye un conjunto de acciones a tomar.
 * Es utilizada por la libreria de planes.
 *
 * @version     0.00    Junio/1997
 * @author      Elohim Fuon
 */

public class Plan extends Vector {

  public Plan() {     // Constructor
    super();
  }

  public void addAction(String a) {
    super.addElement(a);
  }

  public final String actionAt(int i) {
    return((String)super.elementAt(i));
  }

}
```

AgentProperties

```
/* @(#)AgentProperties.java */

package agentes.conocimiento;

import aglet.*;
import java.io.*;
import java.util.Vector;
import java.util.Properties;
import java.util.StringTokenizer;
import agentes.documentos.*;

/**
 * Esta clase constituye un conjunto de datos acerca de
 * las propiedades del agente, asi como las preferencias del
 * usuario que le corresponden. Esta clase permite guardar
 * todos esos datos en disco duro, para leerlos de nuevo
 * mas tarde.
 *
 * @version     0.00    Junio/1997
 * @author      Elohim Fuon
 */
```

```
public class AgentProperties extends Properties {
  private AgletID masterID;       // Identificador del agente superior
  private String fileOnDiskName; // Nombre del archivo de propiedades

  public AgentProperties(AgletID ID, String filename) {     //
Constructor
    super();
    masterID = ID;
    fileOnDiskName = filename;
  }

  public final AgletID getMasterID() {
    return masterID;
  }

  public final void setMasterID(AgletID ID) {
    masterID = ID;
  }

  public final String getFilename() {
    return fileOnDiskName;
  }

  //
  // Regresa el conjunto de keywords del usuario
  //
  public final KeywordSet getUserKeywords() {
    KeywordSet k = new KeywordSet();
    String rt = getProperty("rates");
    String kw = getProperty("keywords");
    if((rt != null) && (kw != null)) {
      StringTokenizer kTokens = new StringTokenizer(kw);
      StringTokenizer rTokens = new StringTokenizer(rt);
      while (kTokens.hasMoreTokens()) {
        try {
          k.addKeyword(new Keyword(kTokens.nextToken(),
                       (new Integer(rTokens.nextToken())).intValue()));
        } catch(Exception e) {}
      }
    }
    return k;
  }

  //
  // Establece el conjunto de keywords del usuario
  //
  public final void setUserKeywords(KeywordSet k) {
    String ks = "";
    String rs = "";
    k.normalize();
```

```
    for(int i=0; i < k.size()-1; i++) {
      ks = ks + k.keywordAt(i).getWord() + " ";
      rs = rs + Integer.toString(k.keywordAt(i).getRate()) + " ";
    }
    if(k.size()>0) {
      ks = ks + k.keywordAt(k.size()-1).getWord();
      rs = rs + Integer.toString(k.keywordAt(k.size()-1).getRate());
    }
    put("keywords", ks);
    put("rates",    rs);
  }

  //
  // Regresa el número de agente con el que debe contar el sistema
  //
  public final int getNumAgents() {
    try {
      return (new Integer(getProperty("numagents"))).intValue();
    } catch(Exception e) {
      return 0;
    }
  }

  public final void setNumAgents(int n) {
    put("numagents", Integer.toString(n));
  }

  public void load() {  // Carga las propiedades desde el archivo en
disco
    try {
      super.load(new FileInputStream(fileOnDiskName + ".properties"));
    } catch(IOException e) {
      e.printStackTrace();
      put("keywords","");
      put("rates", "");
    }
  }

  public void save() {  // Guarda las propiedades en disco
    try {
      super.save(new FileOutputStream(fileOnDiskName + ".properties"),
"Agent Properties");
    } catch(IOException e) {
    }
  }

  //
  // Establece el conjunto de keywords del usuario a partir de los
  // keywords que forman una consulta.
  //
  public final void adjustToQuery(Query q) {
```

```
    setUserKeywords(new KeywordSet(q.toText()));
  }

}
```

PlanLibraryEntry

```
/* @(#)PlanLibraryEntry.java */

package agentes.conocimiento;

/**
 * Esta clase constituye un conjunto de un conjunto de acciones
 * a tomar para conseguir pasar de un estado actual a un estado
deseado,
 * dado un estado determinado del ambiente.
 * Es utilizada por la libreria de planes.
 *
 * @version      0.00    Junio/1997
 * @author       Elohim Puon
 */

public class PlanLibraryEntry {
  private Status currentStatus;    // Estado actual del ambiente
  private Status desiredStatus;    // Estado deseado del agente
  private Plan actions;            // Conjunto de acciones a tomar

  public PlanLibraryEntry(Status c, Status d, Plan a) {   //
Constructor
    currentStatus = c;
    desiredStatus = d;
    actions       = a;
  }

  //
  // Indica si este plan es adecuado dados los estados c y d.
  //
  public final boolean matches(Status c, Status d) {
    return(c.contains(currentStatus) &&
d.isContainedIn(desiredStatus));
  }

  //
  // Indica si este plan ayudaria a conseguir el estado d.
  //
  public final boolean enables(Status d) {
    return d.isContainedIn(desiredStatus);
  }

  public final Plan getPlan() {
```

```
    return actions;
  }

  public final Status getCurrentStatus() {
    return currentStatus;
  }

}
```

DPlanLibrary

```
/* @(#)DPlanLibrary.java */

package agentes.conocimiento;

import agentes.Descubridor;
import java.util.Vector;

/**
 * Clase que constituye una tabla de decisión para el agente
Descubridor.
 *
 * see PlanLibraryEntry.java
 * @version     0.00    Julio/1997
 * @author      Elohim Pinon
 */

public class DPlanLibrary extends Vector {
  Descubridor agent;      // Apuntador al agente Descubridor

  public DPlanLibrary(Descubridor a) {     // Constructor
    super();
    fill();
    agent = a;
  }

  //
  // Inicializa la libreria de planes
  //
  private final void fill() {
    { Status s = new Status();
      Status d = new Status();
      s.put("Nuevo sitio disponible", new Boolean(true));
      d.put("Estoy en un nuevo sitio", new Boolean(true));
      Plan   a = new Plan();
      a.addAction("Viaja a un nuevo sitio");
      addEntry(new PlanLibraryEntry(s, d, a));
    }
    { Status s = new Status();
      Status d = new Status();
```

```
    s.put("Nuevo sitio disponible", new Boolean(false));
    d.put("Estoy en un nuevo sitio", new Boolean(true));
    Plan   a = new Plan();
    a.addAction("Lleva documentos cualesquiera");
    a.addAction("Regresa al hogar");
    addEntry(new PlanLibraryEntry(s, d, a));
  }
  { Status s = new Status();
    Status d = new Status();
    d.put("Llevo suficientes documentos", new Boolean(true));
    Plan   a = new Plan();
    a.addAction("Selecciona documentos");
    addEntry(new PlanLibraryEntry(s, d, a));
  }
  { Status s = new Status();
    Status d = new Status();
    d.put("He sido reconocido", new Boolean(true));
    Plan   a = new Plan();
    addEntry(new PlanLibraryEntry(s, d, a));
  }
  { Status s = new Status();
    Status d = new Status();
    d.put("He enviado nuevos documentos", new Boolean(true));
    Plan   a = new Plan();
    a.addAction("Envia nuevos documentos");
    addEntry(new PlanLibraryEntry(s, d, a));
  }
  { Status s = new Status();
    Status d = new Status();
    d.put("Estoy en el hogar", new Boolean(true));
    Plan   a = new Plan();
    a.addAction("Regresa al hogar");
    addEntry(new PlanLibraryEntry(s, d, a));
  }
}

//
// Añade a la libreria una nueva entrada
//
private final void addEntry(PlanLibraryEntry entry) {
  super.addElement(entry);
}

//
// Indica el plan a seguir para lograr un estado deseado
//
public final Plan getPlanFor(Status ds) {
  Vector choices = new Vector();
  for(int i=0; i<size(); i++) {
    if(((PlanLibraryEntry)elementAt(i)).enables(ds)) {
      choices.addElement((PlanLibraryEntry)elementAt(i));
```

```
        }
      }
      if (choices.size() == 0) return new Plan();
      else if (choices.size() == 1) return
((PlanLibraryEntry)choices.elementAt(0)).getPlan();
      else {
        Status es = new Status();
        for(int i=0; i<choices.size(); i++)

es.add(((PlanLibraryEntry)choices.elementAt(i)).getCurrentStatus());
        es = agent.getEnvironmentStatus(es);  // Pide al agente detalles
sobre la situación.
        for(int i=0; i<choices.size(); i++) {
          PlanLibraryEntry ple = (PlanLibraryEntry)choices.elementAt(i);
          if(ple.matches(es,ds)) {
            return ple.getPlan();
          }
        }
      }
      return new Plan();
    }

}
```

Document

```
/* @(#)Document.java */

package agentes.documentos;

import aglet.*;
import java.io.*;
import java.util.Vector;

/**
 * Esta clase constituye un documento html, y
 * es utilizada por los agentes móviles.
 *
 * @version     0.00    Junio/1997
 * @author      Elohim Puon
 */

public class Document implements Serializable {
  private KeywordSet keywords;
  private int grade;
  private AgletID ownerAgletID;
  private String path;

  public Document(String p) {     // Constructor
    keywords = new KeywordSet();
```

```
    grade = -1;
    path = p;
  }

  public final void qualify(int g) {
    grade = g;
  }

  public final int getGrade() {
    return grade;
  }

  public final void setOwnerAgletID(AgletID id) {
    ownerAgletID = id;
  }

  public final AgletID getOwnerAgletID() {
    return ownerAgletID;
  }

  private final void extractKeywords() throws IOException {
    String   word="";
    int      car, maximum=0;
    boolean  alreadyExists, tag = false;
    FileInputStream stream = new FileInputStream(path);
    keywords.removeAllElements();
    while((car = stream.read()) != -1) {
      car = Character.toLowerCase((char)car);
      if("áéíóúüabcdefghijklmnñopqrstuvwxyz".indexOf(car) < 0) {
        if(word.length()>4) {
          alreadyExists = false;
          for(int i=0; i < keywords.size(); i++)
            if(word.equals(keywords.keywordAt(i).getWord())) {
              alreadyExists = true;
              keywords.keywordAt(i).incTimes();
              if(keywords.keywordAt(i).getTimes() > maximum)
                maximum = keywords.keywordAt(i).getTimes();
              i = keywords.size();
          }
          if(!alreadyExists) {
            Keyword kw = new Keyword(word);
            keywords.addKeyword(kw);
            if(maximum == 0) maximum++;
          }
          if(car == '<') tag = true;
          else if(car == '>') tag=false;
        }
        word="";
      }
      else {
        if(!tag) word=word+(char)car;
```

```
        }
      }
      stream.close();
      for(int i=0; i < keywords.size(); i++) {
        keywords.keywordAt(i).normalize(maximum, 0);
        if(keywords.keywordAt(i).getRate() < 60) {
          keywords.removeElementAt(i--);
        }
      }
    }

    public final KeywordSet getKeywords() throws IOException {
      if(keywords.size()==0) extractKeywords();
      return(keywords);
    }

    public final int getProximity(KeywordSet external) throws
IOException {
      if(keywords.size()==0) extractKeywords();
      return(keywords.getProximity(external));
    }

    public final String getPath() {
      return path;
    }

    public final void setURL(String url) {
      path = url + path;
    }

}
```

DocumentPool

```
/* @(#)DocumentPool.java */

package agentes.documentos;

import aglet.*;
import java.io.*;
import java.util.Vector;

/**
 * Esta clase constituye un conjunto de documentos html, y
 * es utilizada por los agentes móviles e intermediarios.
 *
 * @version     0.00    Junio/1997
 * @author      Elohim Puon
 */
```

```
public class DocumentPool extends Vector {
  private Query query;

  public DocumentPool() {     // Constructor
    super();
    query = new Query("");
  }

  public void addDocument(Document d) {
    addElement(d);
  }

  public boolean removeDocument(String path) {
    for(int i=0; i<size(); i++)
      if(((Document)elementAt(i)).getPath().equals(path))
        return removeElement(elementAt(i));
    return false;
  }

  public final Document documentAt(int i) {
    return((Document)elementAt(i));
  }

  public final int indexOf(String path) {
    for(int i=0; i<size(); i++)
      if(((Document)elementAt(i)).getPath().equals(path))
        return(i);
    return -1;
  }

  public final Document getDocument(String path) {
    try {
      return documentAt(indexOf(path));
    } catch(ArrayIndexOutOfBoundsException e) {
      return null;
    }
  }

  public final DocumentPool addPool(DocumentPool newPool) {
    DocumentPool newDocs = new DocumentPool();
    for(int i=0; i < newPool.size(); i++) {
      Document d = newPool.documentAt(i);
      if(indexOf(d.getPath()) < 0) {
        addDocument(d);
        newDocs.addDocument(d);
      }
    }
    newDocs.setCausalQuery(newPool.getCausalQuery());
    return newDocs;
  }
```

```
  public final void setOwnerAgletID(AgletID id) {
    for(int i=0; i<size(); i++) documentAt(i).setOwnerAgletID(id);
  }

  public final AgletID getOwnerAgletID() {
    if (size() > 0) return documentAt(0).getOwnerAgletID();
    else return null;
  }

  public final void setCausalQuery(Query q) {
    query = q;
  }

  public final Query getCausalQuery() {
    return query;
  }

  public final DocumentPool select(KeywordSet q, int threshold) {
    DocumentPool utilDocs = new DocumentPool();
    for(int i=0; i<size(); i++) {
      try {
        if(documentAt(i).getProximity(q) > threshold) {
          utilDocs.addDocument(documentAt(i));
        }
      } catch (IOException e) {
        System.out.println("No se pudieron extraer los keywords de " +
documentAt(i).getPath());
      }
    }
    utilDocs.setCausalQuery(new Query(q.toText()));
    return utilDocs;
  }

}
```

Keyword

```
/* @(#)Keyword.java */

package agentes.documentos;

import java.io.*;

/**
 * Esta clase constituye un keyword o palabra clave.
 *
 * @version      0.00    Junio/1997
 * @author       Elohim Puon
 */
```

```
public class Keyword implements Serializable {
  private String word;
  private int rate;
  private int times;

  public Keyword(String w) {              // Constructor
    word=w;
    times=1;
  }

  public Keyword(String w, int r) {       // Constructor
    word  = w;
    times = r;
    rate  = r;
  }

  public final String getWord() {
    return word;
  }

  public final int getTimes() {
    return times;
  }

  public final void incTimes() {
    times++;
  }

  public final int getRate() {
    return rate;
  }

  public final void setRate(int r) {
    rate = r;
  }

  public final void normalize(int max, int min) {
    if(min > 0) min = 0;
    try {
      rate=(int)((times-min)*100/(max-min));
    } catch (ArithmeticException e) {
      rate=0;
    }
  }

}
```

KeywordSet

```
/* @(#)KeywordSet.java */
```

```
package agentes.documentos;

import java.util.Vector;
import java.util.StringTokenizer;
import java.io.*;

/**
 * Esta clase constituye un conjunto de documentos html, y
 * es utilizada por los agentes móviles e intermediarios.
 *
 * @version      0.00    Junio/1997
 * @author       Elohim Pion
 */

public class KeywordSet extends Vector implements Serializable {

  public KeywordSet() {     // Constructor
    super();
  }

  public KeywordSet(String ks) {
    StringTokenizer st = new StringTokenizer(ks.toLowerCase());
    while (st.hasMoreTokens()) {
      addKeyword(new Keyword(st.nextToken(), 100));
    }
  }

  public void addKeyword(Keyword k) {
    addElement(k);
  }

  public boolean removeKeyword(Keyword k) {
    return removeElement(k);
  }

  public final Keyword keywordAt(int i) {
    return((Keyword)elementAt(i));
  }

  public final int indexOf(String word) {
    for(int i=0; i<size(); i++)
      if(((Keyword)elementAt(i)).getWord().equals(word))
        return(i);
    return -1;
  }

  public final Keyword getKeyword(String word) {
    try {
      return keywordAt(indexOf(word));
    } catch(ArrayIndexOutOfBoundsException e) {
```

```
      return null;
    }
  }

  public final void addSet(KeywordSet setToAdd, int grade) {
    for(int i=0; i < setToAdd.size(); i++) {
      Keyword k = setToAdd.keywordAt(i);
      int ponderedRate = (int)(k.getRate() * (grade-4) / 50);
      if(indexOf(k.getWord()) < 0) {
        addKeyword(new Keyword(k.getWord(), ponderedRate));
      } else {
        Keyword kw = getKeyword(k.getWord());
        kw.setRate(kw.getRate() + ponderedRate);
      }
    }
    normalize();
  }

  public final void normalize() {
    int max = 0;
    int min = 0;
    for(int i=0; i<size(); i++) {
      int t = keywordAt(i).getTimes();
      if(t > max) max = t;
      if(t < min) min = t;
    }
    for(int i=0; i<size(); i++) {
      Keyword k = keywordAt(i);
      k.normalize(max, min);
      if(k.getRate() < 3) removeKeyword(keywordAt(i--));
    }
  }

  public final String toText() {
    String s = new String();
    for(int i=0; i < size(); i++) s = s + keywordAt(i).getWord() + "
";
    return s;
  }

  public final int getProximity(KeywordSet ks) {
    int prox = 0;
    Keyword dk, uk;
    normalize();
    ks.normalize();
    for(int i=0; i<ks.size(); i++) {
      uk = ks.keywordAt(i);
      dk = getKeyword(uk.getWord());
      if(dk != null) try {
        if(uk.getRate() > dk.getRate()) {
          prox += (int)(100 * dk.getRate() / uk.getRate());
```

```
        } else {
          prox += (int)(100 * uk.getRate() / dk.getRate());
        }
      } catch (ArithmeticException e) {
      }
    }
    if(ks.size() > 0) return(prox / ks.size());
    else return 100;
  }

}
```

Query

```
/* @(#)Query.java */

package agentes.documentos;

import java.util.Date;

/**
 * Esta clase constituye un conjunto de parámetros que definen
 * una consulta del usuario, entre ellos, el conjunto de keywords.
 *
 * @version      0.00    Junio/1997
 * @author       Elchim Puon
 */

public class Query extends KeywordSet {
  private Date timeStamp;

  public Query(String qs) {     // Constructor
    super(qs);
    timeStamp = new Date();
  }

}
```

BIBLIOGRAFIA

Programación

[1] Java Distributed Systems, http ://chatsubo.javasoft.com/current/
[2] Using RMI,
http ://java.sun.com/products/jdk/1.1/docs/guide/rmi/getstart.doc.html
[3] Danny Lange and Mitsuru Oshima, Aglets Cookbook,
http://www.trl.ibm.co.jp/aglets/aglet-book.html
[4] Aglets Mailing List, http://www.trl.ibm.co.jp/aglets/intouch.html
[5] Laura Lemay, Charles L. Perkins, Teach Yourself Java in 21 days, 1996.

Plataformas

[6] Aglets Workbench, http://www.trl.ibm.co.jp/aglets/index.html
[7] Voyager, http ://www.objectspace.com/voyager
[8] Odissey, http ://www.genmagic.com/odissey.html
[9] Java Agent Template, http ://cdr.stanford.edu/ABE/JavaAgent.html
[10] Joseph Kiniry and Daniel Zimmerman, The authors downloaded and evaluated three commercial Java mobile-agent systems. Vol 1, No. 4, Julio/Agosto 1997.
[11] TeleScript, http://www.genmagic.com/Telescript/index.html

Agentes para Internet

[12] Michael N. Huhns and Munindar P. Singh, Internet-Based Agents: Applications and Infrastructure.
http://dlib.computer.org/ic/books/ic1997/pdf/w4008.pdf
Vol 1, No. 4, Julio/Agosto 1997.
[13] Björn Hermans, Intelligent Software Agents on the Internet,
http ://www.hermans.org/agents

Recopilación y Filtrado de Información

[14] Co-operative Information Retrieval in Digital Libraries. Michail Salampasis, John Tait, Chris Bloor, University of Sunderland, UK.
http://osiris.sund.ac.uk/~cs0msa/bcsir96.htm
[15] Information Retrieval and Filtering,
http://agents.www.media.mit.edu/groups/agents/papers/newt-thesis
[16] A Learning Approach to Personalized Information Filtering, Dept. of Electrical Engineering and Computer Science, MIT, 1994.
ftp://info.cern.ch/pub/www/doc/html-spec.ps

Agentes e Información

[17] Donald McKay, Jon Pastor, Rbon McEntire and Tim Finin, An architecture for information agents. AAAI Press, May 1996, http://www.cs.umbc.edu/kqml/papers/arpi.ps
[18] Learning Personal Agents for Text Filtering and Notification. International Conference of Knowledge Based Systems (KBCS 96) , 1996 http://www.cs.cmu.edu/~softagents/papers/kbcs96.ps
[19] Victor R. Lesser, Tim Oates, Cooperative Information Gathering, University of Massachusetts Technical Report, April / 1994.
[20] Keith Decker, Victor Lesser, MACRON : An Architecture for Multiagent Cooperative Information Gathering, University of Massachusetts, Technical Report, November / 1994.

Sistemas Multiagentes

[21] Russell, S., and P. Norvig, Artificial Intelligence: A Modern Approach, 1995
[22] O'Hare and N.R. Jennings, Foundations of Distributed Artificial Intelligence, 1996, Sixth-Generation Computer Technology Series.
[23] Edmund H. Durfee, Jeffrey S. Rosenschein, Distributed Problem Solving and [24] Multi-Agent Systems : Comparisons and Examples. AI Press, 1995.

[25] JyiShane Liu, Katia Sycara,Distributed Problem Solving through Coordination in a Society of Agents, 1995. Robotics Institute, Carnegie Mellon University.
[26] Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents, http://www.msci.memphis.edu/~franklin/AgentProg.html
[27] Why Multiagent Systems?
http://www.cs.cmu.edu/afs/cs/usr/pstone/public/papers/96ieee-survey/node5.
[28] Susan E. Lander and Victor R. Lesser, Customizing Distributed Search among Agents, University of Massachusetts Technical Report, 1995.
[29] KQML, Knowledge Query and Manipulation Language
http://retriever.cs.umbc.edu/kqml/
[30] Georgeff, M. P., and Ingrand, F.F. Decision making in an embedded reasoning system. Proc. Int. Jt. Conf. Artif. Intell, 11th, 1989.