Sistema de Comunicación Orientado a Organizaciones de Agentes

Juanmi Alberola

GTI-IA

Septiembre 2008

Dirigido por: Ana García-Fornes
Agustín Espinosa

DSIC

Índice general

Introducción

Los sistemas basados en agentes son una de las áreas más interesantes que han surgido en los últimos años desde el punto de vista de la tecnología de la información. El concepto de agente inteligente está presente en muchas disciplinas de esta tecnología como pueden ser la ingeniería del software, redes de computadores, inteligencia artificial, sistemas distribuídos, sistemas móviles, sistemas de control, telemáticos, comerio electrónico, etc.

Desde la aparación de la tecnología de agentes, los investigadores han hecho hincapié en los beneficios que ésta nos ofrece, las particularidades que podríamos extraer de ella por la aproximación que tiene al comportamiento humano y en definitiva, la nueva visión orientada a la programación con agentes que nos proporciona este paradigma.

Una de las áreas de investigación dentro de esta tecnología, en la que diversos grupos han invertido grandes esfuerzos, se centra en el diseño y desarrollo de Plataformas Multiagente que den soporte a los Sistemas Multiagente (SMA). Existen multitud de estas plataformas, cada una de ellas ofrece una serie de funcionalidades, tienen unas arquitecturas internas y utilizan una tecnología de comunicación entre agentes.

La tecnología de agentes requiere un soporte para desarrollar aplicaciones abiertas, complejas y a gran escala. En estos entornos, la escalabilidad, eficiencia y robustez de las plataformas multiagentes (MAP) que ejecutan estas aplicaciones son características esenciales. Como se comenta en [9], la comunidad de sistemas multiagentes debería diseñar e implementar sistemas grandes, compuestos por cientos de agentes y no sólo por unos pocos. Para desarrollar estos sistemas, los investigadores necesitan disponer de plataformas eficientes y escalables. Los actuales diseños no están generalmente pensados para obtener un alto rendimiento de la plataforma. Actualmente, en muchos diseños, la funcionalidad ofrecida y la facilidad de desarrollo son características que se tienen más encuenta que la eficiencia. Muchos desarrolladores, por ejemplo, desarrollan su propio prototipado de plataforma para probar algunos de los conceptos sobre los que investigan, como teorías de

razonamiento, sistemas normativos, argumentación, etc. sin tener en cuenta los requisitos de posibles aplicaciones reales que se desarrollen sobre dicha plataforma.

Es por ello que muchas de las plataformas actuales, no son adecuadas para la ejecución de sistemas complejos, porque sus diseños no están realizados para maximizar características como la eficiencia y la escalabilidad. Los estudios realizados en [17], [9], [25], [10], [8], [7], [21], [15], [20], [22], [14], [19], [13], [6] evalúan varios aspectos de las plataformas más populares, entre ellos el rendimiento del mecanismo de comunicación de agentes ofrecido. De todos estos estudios se concluye que las plataformas actuales presentan, en mayor o menor medida, una degradación en el tiempo de respuesta cuando la carga de la misma es alta. Además, cuando se ejecutan sistemas multiagente a gran escala, el rendimiento de muchas plataformas se decrementa considerablemente, incluso algunas plataformas fallan [24].

De esta forma, si los SMA están pensados para abarcar un gran número de ordenadores, distribuídos en localizaciones posiblemente separadas y que contengan un volumen considerable de agentes que se ejecutan produciendo un alto grado de interacciones entre ellos, necesitamos disponer de plataformas que nos ofrezcan un alto grado de robustez, escalabilidad y eficiencia.

Este tipo de sistemas de gran escala, compuestos por muchos ordenadores, están íntimamente relacionados con los sistemas abiertos. En este sentido, las organizaciones de agentes constituyen una área emergente de los SMA, que depende de conceptos como sistemas abiertos y heterogéneos. Los diseños organizacionales juegan un papel clave en el diseño de SMA abiertos y complejos. Cuando los sistemas pasan a estar compuestos por miles de agentes, tenemos que cambiar la perspectiva de un agente individual de coordinación y control a una perspectiva organizacional. Las aplicaciones de agentes

Las organizaciones representan una potente herramienta para coordinar el comportamiento de una sociedad de individuos ([11]). Las organizaciones humanas han sido estudiadas en una variedad de ramas de la ciencia como la psicología, economía, sociología, etc. Es importante la coordinación para prevenir el caos de una sociedad de agentes, ya que tienen objetivos individuales que pueden interferir con otros. También lo es, para llevar a cabo objetivos globales que no pueden ser resueltos a nivel individual, así como para compartir información y conocimiento entre los agentes ([11], [18])

Para poder usar el concepto de organización, como un elemento para desarrollar SMA, es necesario que la plataforma dónde se está ejecutando el SMA ofrezca las características necesarias para definir organizaciones de agentes: crear agrupaciones de agentes, definir objetivos globales, asociar tareas o planes a agentes individuales, asociar roles a agentes y expresar las relaciones o protocolos de interacción entre los mismos, representar un conjunto de reglas o restricciones del sistema, etc. Además por supuesto, de ofrecer unas características aceptables de eficiencia y escalabilidad, ya que, como se ha dicho, el concepto de organización está relacionado con el de sistemas multiagentes de gran volumen y abiertos.

Agent organizations are an emergent area of MAS that relies on the notion of openness and heterogeneity of MAS and poses new demands on traditional MAS models.

These demands include the integration of organizational and individual perspectives and the dynamic adaptation of models to organizational and environmental changes. . Organizational self-design will play a critical role in the development of larger and more complex MAS. As systems grow to include hundreds or thousands of agents, we must move from an agent-centric view of coordination and control to an organization-centric one. Practical applications of agents to organizational modeling are being widely developed but formal theories are needed to describe interaction and organizational structure. Furthermore, it is necessary to get a closer look at the relation between organizational roles and the agents that fulfil them.

Un sistema multiagente complejo, donde hay un gran número de agentes, puede tener cierta similitud a las organizaciones humanas. Estas organizaciones definen unos roles asociados a los individuos, unos objetivos a cumplir por la sociedad, unos planes o estrategias para llegar a esos objetivos, y unas normas o reglas que hay que cumplir.

El concepto de una organización en un sistema multiagente no está comunmente definido por la comunidad tecnológica. Sin embargo, aunque conceptualmente haya algunas diferencias, es importante destacar que palabras como agrupación, coordinación, cooperación, etc., están íntimamente relacionadas con el concepto de organización de agentes. Filosóficamente hay una gran discusión entre lo que sería una organización desde el punto de vista de sistemas multiagente: para algunos puede ser algo dinámico mientras que para otros es un concepto global del sistema multiagente. En cualquier caso, la necesidad de agrupar y coordinar distintos comportamientos de los agentes es necesario para representar sistemas, especialmente complejos que requieran de cierto control y cooperación para lograr objetivos. El concepto de organización de agentes nos ofrece un entorno que nos ayuda a simplificar, estructurar y desarrollar de una manera más flexible a la vez que potente, los SMA.

Motivación

Tal y como hemos comentado en la introducción , la eficiencia y escalabiliad son dos características importantes que deberían estar presentes en las plataformas multiagente, especialmente cuando se están ejecutando sistemas complejos, con un gran número de agentes e interacciones entre ellos. En estudios anteriores [17] se han podido analizar aspectos internos de las plataformas actuales que hacen que baje eficiencia de las mismas y el rendimiento sea cada vez más pobre, según va aumentando la complejidad del sistema.

Muchas de las plataformas actuales no están especialmente preparadas para la ejecución de sistemas multiagente complejos. De hecho, no hay estudios empíricos de pruebas de rendimiento para las distintas plataformas con un gran número de carga. La mayoría de los estudios de rendimiento se centran en sistemas formados por pocos agentes y que se están ejecutando en pocas máquinas, no más de diez. Nuestra experiencia utilizando los benchmarks planteados en [24] nos ha permitido averiguar que la mayoría de las plataformas actuales se degradan considerablemente cuando la magnitud del SMA que se está ejecutando aumenta, incluso, algunas de ellas, directamente se caen.

Las plataformas actuales están basadas en lenguajes interpretados como Java o Python. Aunque estos lenguajes ofrecen algunas ventajas como portabilidad y un fácil desarrollo, las plataformas que están desarrolladas en estos lenguajes no ofrecen un alto rendimiento, especialmente cuando están ejecutando sistemas complejos.

Este comportamiento puede estar influenciado por los middlewares que se requieren para interpretar estos lenguajes (como la JVM). Estudios previos nos han permitido llegar a la conclusión que el hecho de utilizar los enfoques actuales para desarrollar plataformas multiagente, puede afectar al rendimiento del sistema que se está ejecutando [5]. Utilizar los servicios ofrecidos por el sistema operativo para desarrollar la plataforma en lugar de usar un middleware entre el sistema operativo y la plataforma, puede mejorar notablemente el rendimiento y la escalabilidad de la plataforma.

Por lo que a nosotros se refiere, la funcionalidad de una plataforma multiagente puede ser vista como una extensión de la funcionalidad ofrecida por el sistema operativo. Así, la robustez y la eficiencia de una plataforma multiagente debería ser requerida, de la misma manera que requerimos la robustez y la eficiencia en un sistema operativo.

En nuestra opinión, desarrollar plataformas multiagente más cercanas al sistema operativo mejorará el rendimiento de plataformas complexas, abiertas y a gran escala, debido en parte, a que no se requiere ningún middleware como la JVM. Este diseño puede incrementar la robustez, eficiencia y la escalabilidad.

En vista de las carencias ofrecidas por las plataformas actuales en sistemas complejos, en este trabajo se propone un nuevo enfoque para desarrollar sistemas multiagente a través de una plataforma más cercana al sistema operativo, teniendo en cuenta los estudios anteriores [17], [24] y los diseños de las arquitecturas internas de algunas plataformas ([4], [23], [16]). Esta plataforma no sólo ofrecerá la funcionalidad requerida para desarrollar sistemas multiagente, sinó que además, estará orientada a ofrecer un alto rendimiento, aunque estemos ejecutando sistemas a gran escala, compuestos por miles de agentes que se ejecuten en muchos hosts. Escenarios donde, la eficiencia y la escalabilidad son características especialmente requeridas.

Como hemos comentado, el hecho de hablar de sistemas complejos y abiertos, nos relaciona directamente las organizaciones de agentes. Las organizaciones de agentes son un instrumento potente que permite la estructuración, coordinación y cooperación de los mismos para llevar a cabo unos objetivos. Este concepto es importante especialmente en los entornos que estamos hablando, de sistemas complejos con gran número de agentes e interacciones entre ellos. Definiendo unos roles y unas interacciones entre ellos, se pueden especificar tareas o estrategias asociadas a los mismos para lograr unos objetivos globales, siguiendo unas normas o reglas de comportamiento que deben cumplir los agentes de la organización.

Las organizaciones se han tratado en varias áreas de la ciencia y actualmente, en los sistemas multiagente está siendo ampliamente estudiada. Conceptos relacionados de una u otra forma con las organizaciones de agentes como sistemas abiertos, negociación, argumentación, contratación, reputación, confianza, conocimiento distribuído, etc. están siendo aplicados al área de los SMA y están en contínuo proceso de innovación e investigación. Es por tanto, un concepto vivo y que está siendo tratado y estudiado por muchos grupos de investigación del área.

El concepto de organización de agentes se convierte en un requisito importante que debería ofrecer una plataforma que esté pensada para ejecutar aplicaciones de un volumen considerable, ya que nos ofrece un entorno que nos ayuda a simplificar, estructurar y desarrollar de una manera más flexible a la vez que potente, los SMA. Muchos grupos de investigación del área de sistemas multiagente, basan sus estudios en definir propuestas teóricas así como metodologías para implementar sociedades de agentes. Y para dar soporte a la ejecución de todas estas propuestas teóricas, necesitamos entornos como pueden ser las plataformas. Lamentablemente, en la literatura, no existen muchos entornos que incorporen de alguna u otra forma conceptos relacionados con las organizaciones, sólo

pocas plataformas ofrecen algunas de las características para poder implementar sistemas multiagentes de estas características, entre ellas Jack, Madkit, Spade o Zeus son algunos ejemplos. Dadas las características de Magentix como la eficiencia y la escalabilidad, esta plataforma sería idónea para incorporar el concepto de organizaciones de agentes.

La propuesta principal de este trabajo se centra en diseñar un soporte para la integración del concepto de organización de agentes en la plataforma Magentix. Primeramente se requerirá un estudio del estado del arte de las organizaciones de agentes. Viendo qué conceptos son necesarios incluir y cómo son incorporados dichos conceptos en ciertas plataformas. Teniendo en cuenta este objetivo, presentaremos un sistema de comunicación para soportar organziaciones de agentes, así como otros de los requisitos de éstas. En las siguientes secciones se planteará el diseño inicial de esta plataforma y se discutirá cómo se pueden aprovechar algunas de las características de la misma para ofrecer soporte a organizaciones de agentes. Se han dejado al margen características de las organizaciones de agentes como la definición de objetivos globales así como toda la parte normativa, de manera que quede abierta una línea de trabajo futuro para continuar ampliando la funcionalidad de la plataforma, en la dirección de organizaciones de agentes.

Plataforma Magentix

Introducción

La plataforma MAGENTIX ([2]) está desarrollada en C sobre el Sistema Operativo (SO) Linux. Se trata de una plataforma para dar soporte a sistemas multiagente de manera distribuída, por tanto, se puede ejecutar sobre diversos ordenadores, manteniendo la información necesaria de forma replicada.

Para la construcción de esta plataforma, se han tenido en cuenta conceptos relacionados en diversas áreas de la computación, como son la inteligencia artificial, los sistemas operativos y los sistemas distribuídos.

La plataforma, estará formada por un conjunto de ordenadores ejecutando el SO Linux. En cada uno de ellos existirá un proceso llamado *magentix* que se encargará de gestionar la pertenencia de cada ordenador a la plataforma, además de encargarse de la inicialización y finalización ordenada de la parte de la plataforma que reside en ese ordenador.

Magentix ofrece por cada ordenador, una serie de servicios y funcionalidades para dar soporte a los agentes que se están ejecutando en dicho ordenador. Los servicios que hay implementados hasta el momento son los llamados *AMS* (Agent Management Service), el *DF* (Directory Facilitator) y el servicio de mensajería entre agentes (MTS).

Los agentes se representan en Magentix como procesos del SO, y realizarán las comunicaciones entre cada par de agentes mediante conexiones punto a punto, utilizando sockets TCP. Por tanto cada agente de la plataforma tendrá asociado un puerto TCP.

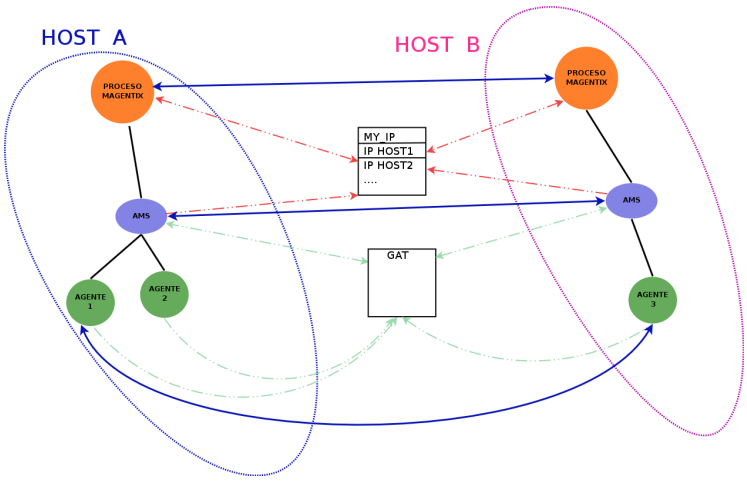

Figura 1: Estructura de la plataforma

Estructura de cada nodo de la Plataforma

La plataforma Magentix ha sido desarrollada basada en los recursos proporcionados por el SO. Su principal objetivo consiste en ofrecer los mismos servicios presentes en la mayoría de plataformas multiagente, pero con unas prestaciones próximas a la cota inferior de tiempo, que se consigue al usar directamente los servicios del SO. La plataforma construida se denomina MAGENTIX (una plataforma Multi-AGENTe integrada en LINUX), y ha sido implementada en el lenguaje C sobre el SO Linux.

La plataforma está formada por un conjunto de ordenadores ejecutando el SO Linux (figura 1). A su vez, Magentix presenta una estructura a nivel de host que puede ser vista como un árbol de procesos. Aprovechando la estructura jerárquica en la que se gestionan los procesos en el SO Linux, y utilizando los conceptos proporcionados este SO como son el envío de señales, la memoria compartida, los hilos de ejecución, los sockets, etc. hemos conseguido disponer de un escenario idóneo para el desarrollo de una plataforma multiagente robusta, eficiente y escalable.

Si observamos la figura 2 podemos ver la estructura de la plataforma Magentix en cada uno de los hosts que componen la plataforma. Observamos que esta estructura está formada por tres niveles de procesos. En el nivel superior se encuentra el proceso *magentix*. Este proceso será el primer proceso creado al lanzar la plataforma en dicho host mediante la órden:

```
mgx start-platform
```

Este comando arrancará una plataforma Magentix en el host donde ha sido lanzada.

Por debajo de este nivel de procesos, encontramos una serie de servicios de la plataforma, entre los cuales podemos referenciar por ejemplo el servicio *AMS* o el *DF* (definidos por FIPA) que podemos ver en la figura 2. Estos servicios darán soporte a los agentes

que se estén ejecutando en dicho host. Como se puede observar, el proceso *magentix* es el proceso padre de todos los servicios que se ejecutan en cada host.

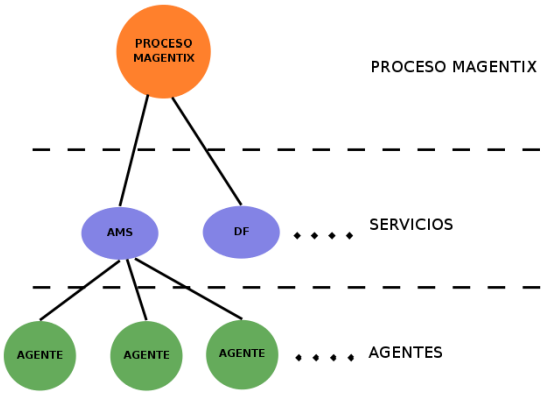

Figura 2: Árbol de procesos

Estos servicios están representados por diferentes procesos (*ams*, *df*, etc.) distribuídos entre los diferentes hosts y que manejan información replicada entre los procesos del mismo servicio. Utilizando las órdenes de *fork* y *exec* proporcionadas por el SO Linux, podemos gestionar la creación de dichos servicios. Además, el hecho de tenerlos estructurados jerárquicamente nos permitirá que el proceso *magentix* pueda realizar una gestión completa sobre los servicios.

El concepto de envío y recepción de señales entre el proceso *magentix* y los procesos servicios de la plataforma, nos permite realizar una inicialización y finalización controladas y ordenadas de la plataforma, así como detectar cualquier caída de un servicio. El proceso *magentix*, al actuar como proceso padre de los servicios, puede enviar la señal *SIGTERM* para matar a los procesos correspondientes a los servicios. Del mismo modo, cuando un proceso hijo muere, envía una señal *SIGCHLD* a su proceso padre. Así, el proceso *magentix* capturará esta señal cada vez que un proceso hijo (los procesos de los servicios) muera.

Finalmente, en un tercer nivel de procesos están los agentes. Cada agente en la plataforma Magentix vendrá representado por un proceso diferente y serán procesos hijos del proceso *ams* que se esté ejecutando en su mismo host. En los siguientes apartados se profundizará en la relación de los agentes con dicho servicio.

Arquitectura Distribuída de la Plataforma

Una vez hemos visto cómo se organizan los componentes de la plataforma dentro de un mismo host, pasaremos a detallar cómo se sincronizan los diferentes componentes de los distintos hosts que forman la plataforma.

La arquitectura distribuída de Magentix nos permite tener información replicada en todos los hosts para lograr mayor eficiencia. Cada host que forma parte de la plataforma,

tiene el conocimiento de todos los hosts que la componen, intercomunicando así, cada par de hosts. Este hecho nos permitirá gestionar correctamente la distinta información que va variando en la plataforma, como veremos más adelante. Para sincronizar la información replicada entre todos los hosts que forman la plataforma dispondremos de un host que será denominado *host principal* y que está diferenciado del resto de hosts. El host principal será el primer host que será informado de cualquier cambio que se produzca, y será el encargado de realizar un broadcast a todos los hosts de la plataforma para informarles de dicho cambio. En cuanto a funcionalidad, todos los hosts de la plataforma tienen la misma funcionalidad que el host principal, ya que en todos se ejecutan los mismos componentes.

La asignación del host principal no es aleatoria. El host principal será todo aquél host que esté ejecutando un proceso *magentix* (con los servicios obligatorios que se lanzan automáticamente) que haya sido lanzado mediante el comando:

```
mgx start-platform
```

Este host será el primero que formará parte de la plataforma. Seguidamente, todos los hosts que queramos unir a la plataforma, ejecutarán también un proceso *magentix* y los servicios obligatorios, pero este proceso *magentix* será lanzado mediante el comando:

```
mgx start-host nombre_del_host
```

Con estas órdenes, especificaremos qué host es el host principal de la plataforma, y por tanto, lo que haremos será unirnos a esa plataforma que se está ejecutando.

Cada proceso *magentix* dispone de una tabla en memoria donde almacena la dirección IP de cada host que pertenece a la plataforma. Esta tabla se encuentra replicada en todos los hosts de la plataforma, así, cada host conocerá qué hosts están en un momento dado, formando parte de la plataforma.

El proceso *magentix* así como los procesos asociados a los servicios, disponen cada uno de ellos, un puerto bien conocido por el que comunicarse mediante sockets TCP. Así por ejemplo, el proceso *magentix* de cada host tiene asociado el puerto *60000*, el servicio *AMS* dispone para sus procesos del puerto *60010*, y así para todos los servicios implementados. Estos puertos nos servirán para comunicar procesos punto a punto. Ya sea entre el proceso *magentix* y un proceso referente a algún servicio del mismo host, o entre dos procesos del mismo servicio de hosts diferentes.

De esta manera, cada proceso *magentix* se puede comunicar con el resto de procesos *magentix* de la plataforma, porque todos tienen asociado el mismo puerto bien conocido de cada una de las diferentes máquinas. Esto nos servirá para sincronizar la información replicada que utilizan estos procesos, cada vez que haya algún cambio. De esta manera, siempre que se una un host a la plataforma o se elimine un host de la plataforma, el host principal, que será el primero en apreciar este cambio, enviará un broadcast al resto de los procesos *magentix* de la plataforma para que actualicen la información de sus respectivas

tablas. La comunicación entre los diferentes procesos *magentix* se realiza mediante el envío de comandos por el puerto bien conocido que tiene asociado este proceso.

Cada proceso *magentix* tiene un hilo que está continuamente escuchando comandos entrantes. Existen una serie de comandos que puede recibir un proceso *magentix*: comando para añadir un host, comando para eliminar un host, y comando para terminar la ejecución de la plataforma en ese host. Si el thread que está escuchando los comandos por el puerto bien conocido recibe un comando, lo procesará. En el caso de recibir un comando para añadir un host o eliminar un host, insertará el nuevo host o borrará el host solicitado de su tabla. El host principal, como hemos comentado, además de modificar su tabla, enviará broadcastings al resto de los hosts para que actualicen su información.

Cuando un nuevo host se une a la plataforma, el proceso *magentix* del nuevo host, enviará un comando al proceso *magentix* del host principal para informarle de la existencia de este nuevo host. Una vez modificada la información de su tabla, el proceso *magentix* del host principal enviará un comando al resto de los hosts de la plataforma para que actualicen su información. De una forma similar se actuaría cada vez que un host termina su ejecución, ya sea voluntariamente o ya sea porque se ha caído. Un caso particular y contemplado en Magentix es la terminación ordenada de la plataforma. Esto es, que cuando el proceso *magentix* del host principal recibe la señal *SIGTERM*, enviará un comando al resto de procesos *magentix* de la plataforma para que terminen su ejecución.

Con este modelo, conseguimos sincronizar todos los hosts pertenecientes a la plataforma, utilizando el proceso *magentix* del host principal como broadcaster, tal y como vemos en la figura 3.

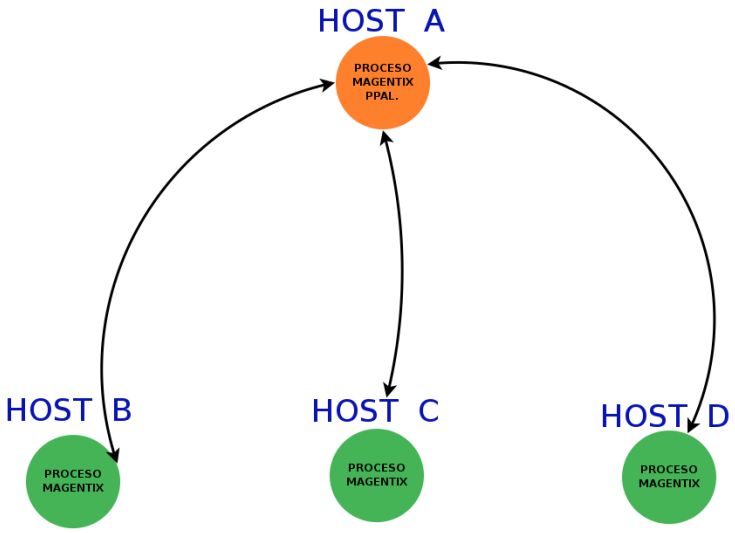

Figura 3: Distribución de los procesos *magentix*

Como veremos más adelante, los servicios de la plataforma también necesitarán sincronizar su información para mantenerla válida. Tal y como ocurre con los procesos *magentix*, es importante que los procesos referentes a los servicios implementados en Ma-

gentix, también conozcan los hosts que componen la plataforma. El hecho de tener estos servicios distribuídos nos introduce esta necesidad para que se pueda sincronizar la información de los distintos hosts. Como hemos explicado anteriormente, los procesos de un mismo servicio tendrán un puerto bien conocido, pero es necesario conocer las distintas IP's que componen los hosts de la plataforma. Esta información es la que reside en la tabla ya comentada de cada proceso *magentix*.

Utilizando el concepto de memoria compartida que nos ofrece Linux podemos conseguir esto. Esta tabla donde se almacenan las direcciones IP's de los distintos hosts de la plataforma, está mapeada en memoria compartida para que los servicios puedan acceder a esta información. Así, cada vez que un servicio necesita conocer todos los hosts que forman parte de la plataforma para actualizar la información que se encuentra replicada en todos los hosts, no se requiere solicitar esa información al proceso *magentix*. Simplemente, el proceso correspondiente, tendrá que acceder a la tabla y consultarla, ya que está mapeada como memoria compartida. Por supuesto, en esta tabla sólo tendrá acceso de escritura el proceso *magentix* de ese host, ya que los servicios simplemente necesitarán consultar esa información.

Servicios de la plataforma

El servicio AMS (Agent Management System)

El servicio *AMS* ofrece la funcionalidad de *páginas blancas*, manteniendo información sobre todos los agentes que se ejectuan en la plataforma. Este servicio se encuentra distribuído en todos los ordenadores que forman parte de la plataforma. En cada uno de ellos, existe un proceso llamado *ams* creado y controlado por el proceso *magentix* tal y como hemos comentado en la sección . Los distintos procesos *ams* comparten información replicada para ofrecer dicho servicio. La funcionalidad principal de este servicio será gestionar los agentes de la plataforma que se han lanzado en ese ordenador.

Tal y como hemos dicho en la sección , todos los agentes que se lancen en un host serán procesos hijos del proceso *ams* que se esté ejectuando en dicho host. Al igual que ocurre entre el proceso *magentix* y los procesos de los servicios, el proceso *ams* tendrá el control total sobre los agentes que se encuentren ejecutando en ese host determinado, ya que es su proceso padre. De esta manera, cada vez que un agente empiece o termine su ejecución, el proceso *ams* podrá gestionar rápidamente estos eventos, ya que es el proceso que realizará la llamada *fork* y el que podrá capturar la señal *SIGCHLD* cuando un proceso hijo muera.

El servicio *AMS* contiene la información de todos los agentes que forman la plataforma. Permite entre otras cosas, a partir de un nombre de agente, encontrar su dirección de contacto para que otros agentes puedan comunicarse con él. Aparte de la información de contacto de todos los agentes de la plataforma, el servicio *AMS* también dispone de información adicional de cada agente como puede ser el identificador del proceso co-

rrespondiente, o el usuario propietario de este proceso. El hecho de tener este servicio distribuído en la plataforma, implica que las consultas sean más eficientes, y que este servicio no sea un cuello de botella, puesto que las operaciones de consultar información de otros agentes son relativamente frecuentes en los sistemas multiagentes.

Como se ha comentado, este servicio viene implementado por los distintos procesos *ams* que están ejecutándose en cada uno de los hosts de la plataforma. Se ha diseñado el servicio de manera que todos los procesos *ams* de la plataforma tengan la información de todos los agentes. Por tanto, estos procesos necesitan sincronizar su información, de manera que cuando un proceso *ams* detecte algún cambio relevante sobre información referente a los agentes que están en su mismo host (y que son procesos hijos suyos), transmitirá esta información al resto de procesos *ams* de la plataforma para que actualicen su información. Para conocer cuáles son los procesos *ams* que forman parte de la plataforma en un momento dado, los procesos *ams* pueden acceder a la tabla que gestiona el proceso *magentix* ya que está mapeada en memoria compartida para que sea leíble por los servicios y además, está replicada en cada host.

Cada proceso *ams* almacena la información referente a los agentes de la plataforma en dos tipos de tablas:

- En la tabla **GAT** (Global Agent Table) se almacena el nombre y dirección de contacto de cada uno de los agentes que forman parte de la plataforma. Como hemos introducido en la Sección , cada agente dispondrá de un puerto TCP para comunicarse con otros agentes. Por tanto, la información de contacto almacenada en esta tabla es la dirección IP y el puerto TCP asociado a cada agente. Esta tabla está implementada como una tabla hash indexada por el nombre de los agentes, de manera que proporcionando un nombre, las operaciones de búsqueda, inserciones y borrados sobre esta tabla tienen un coste promedio constante. En esta tabla está la información de contacto de todos los agentes de la plataforma, por tanto, está replicada en ca, de manera que, consultando esta tabla que se encuentra replicada para cada proceso *ams* de la plataforma, podemos comunicarnos con cualquier agente de la plataforma.

- En la tabla **LAT** (Local Agent Table) se almacena información adicional de los agentes como el propietario, el PID del proceso que representa a este agente o el estado de su ciclo de vida. Esta tabla no está replicada, de manera que cada proceso *ams* tiene en su tabla *LAT*, únicamente la información de los agentes que están en su host (y que son procesos hijos suyos). Al igual que la tabla anterior, la *LAT* se implementa como una tabla hash indexada por el nombre de los agentes.

Siguiendo este esquema de tablas vemos que la información contenida en la *GAT* tiene que estar replicada en todos los hosts. Para ello, procedemos de una forma similar a la explicada anteriormente para los procesos *magentix* (Sección). Cuando un proceso *ams* realiza un cambio en su *GAT* sobre un agente que está bajo su gestión, comunicará este cambio al proceso *ams* del host principal, y éste hará un broadcasting al resto de procesos *ams* comunicándoles el cambio. Cada proceso *ams* tiene un hilo interno de ejecución que

está escuchando un puerto bien conocido para la recepción de comandos. Estos comandos son para añadir un nuevo agente a la GAT, borrar un agente de la GAT, etc. Cuando este hilo reciba un comando lo procesará, de igual manera que procesa comandos el proceso *magentix* explicado anteriormente en la Sección .

El hecho de que cada proceso *ams* sea el padre de cada agente que se encuentre ejecutándose en ese host, hará que las operaciones de inserción y borrado de las tablas sean automáticas para estos agentes. Cuando un proceso *ams* lanza un agente a ejecución, insertará la información necesaria en sus tablas y notificará el cambio al *ams* del host principal para que el resto de hosts incluyan la información en sus respectivas GATs. Cuando un proceso *ams* detecte que un agente ha muerto (recibiendo la señal *SIGCHLD*), podrá borrar la información de sus tablas e informar al *ams* del host principal.

Cada vez que queramos ejecutar un agente en un host que forme parte de la plataforma, ejecutaremos el siguiente comando:

```
mgx start-agent Nombre_del_agente Ruta_del_ejecutable Parámetros
```

Este comando, entre otras cosas enviará un comando al proceso *ams* local para solicitar la creación de un nuevo agente. El proceso *ams*, una vez haya recibido este comando, realizará las llamadas correspondientes a las funciones *fork* y *exec* para crear el nuevo proceso solicitado. Como el proceso *ams* es el padre del proceso que representa al agente que se acaba de lanzar, tendrá toda su información para incluirla en su tabla *LAT* y en su tabla *GAT*. Además, una vez modificadas las entradas correspondientes en sus tablas locales, enviará esta información actualizada al proceso *ams* del host principal, que hará un broadcasting al resto de procesos *ams* de la plataforma para que actualicen su tabla GAT. Por último, cuando un host se una a la plataforma, se le enviará un comando especial al *ams* del host principal para que transfiera todos los datos contenidos en la *GAT* al nuevo proceso *ams* del host que se acaba de unir, de esta manera, la información estará actualizada también en este nuevo host.

La tabla *GAT* se encuentra mapeada en memoria compartida para los agentes. De esta forma, cuando un agente quiere comunicarse con otro y requiera conocer la dirección IP y el puerto TCP, no tendrá ni que realizar una consulta al proceso *ams* local, simplemente leerá la información de la tabla *GAT*. Evidentemente, la tabla será de sólo lectura para los procesos correspondientes a los agentes y de lectura/escritura para los procesos *ams*. Con este modelo, se consigue gran flexibilidad para encontrar la información de contacto de cualquier agente de la plataforma. La replicación de la tabla *GAT* en cada host, así como el mapeo de esta tabla en memoria compartida, permite una mayor eficiencia a la hora de realizar consultas para comunicarse con otros agentes.

Por su parte, la tabla *LAT*, no está replicada ni está mapeada en memoria compartida para los agentes. Este hecho es debido a que se trata de una tabla donde se almacena información que sólo puede ser necesaria para el proceso *ams* del mismo host, pero nunca por agentes. Por lo tanto, no necesita estar replicada en cada host.

El Servicio DF (Directorio Facilitador)

El *DF* ofrece la funcionalidad de *páginas amarillas*, manteniendo información de los servicios ofrecidos por agentes que pertenecen a la plataforma. El *DF*, permite a los agentes poder registrar servicios ofrecidos por ellos, desregistrarlos o realizar búsquedas de servicios ofrecidos por otros agentes y que le sean requeridos. Al igual que el *AMS*, este servicio tiene una funcionalidad distribuída, de manera que en cada ordenador existe un proceso llamado *df* creado y controlado por el proceso *magentix*. La información almacenada está replicada en todos los procesos *df* de la plataforma. De una manera similar a la del servicio *AMS*, cada proceso *df* lanzado en cada host tiene asociado un puerto bien conocido para facilitar la comunicación con los diferentes procesos que implementan este servicio. Además, cada proceso *df* tiene acceso a la tabla de direcciones IP que mapea en memoria compartida su respectivo proceso *magentix* situado en el mismo host.

Un proceso *df* dispone de una tabla llamada **GST** (Global Service Table) para guardar su información. Esta tabla contiene un listado de los servicios ofrecidos por los distintos agentes de la plataforma, junto con el nombre del agente que ofrece dicho servicio. Su implementación se realiza mediante una tabla hash indexada por el nombre del servicio, ya que la manera de proceder en este caso será la búsqueda, inserción y borrado de servicios ofrecidos por los agentes. Esta tabla se encuentra replicada en cada host de la plataforma para cada proceso *df*.

La manera de actualizar la información en todos los hosts es similar a la utilizada para el caso de la GAT explicada en la sección . La comunicación entre los distintos procesos *df* situados en ordenadores diferentes es una comunicación punto a punto. Cada proceso *df* tiene un hilo interno que está escuchando por el puerto bien conocido para recibir comandos. Estos comandos serán para añadir o eliminar servicios. Cuando un proceso *df* añada o elimine un servicio, notificará este cambio al *df* situado en el host principal. Este proceso *df* será el encargado de hacer un broadcasting al resto de procesos *df* para mantener la información sincronizada. Al igual que ocurre en el servicio AMS, cuando un host se une a la plataforma, el *df* de este nuevo host necesita tener la información completa que se contiene en la tabla distribuída GST. El encargado de transferir toda esta información al nuevo *df* será el proceso *df* del host principal.

Los agentes pueden registrar o desregistrar servicios ofrecidos al resto de agentes. Este listado lo mantiene el servicio *DF* de la plataforma mediante la tabla GST replicada y asociada a cada proceso *df* de la plataforma. A diferencia de cómo está implementada la tabla GAT , esta tabla no se encuentra mapeada en memoria compartida, de manera que cuando un agente requiera cualquier operación sobre la GST, necesitará interactuar con el proceso *df* de su mismo host. El proceso *df* por tanto, dispone de una interfaz de agente para poder comunicarse con los agentes.

Se ha desarrollado una API para las operaciones relacionadas con el directorio de servicios: registrar un servicio, desregistrar un servicio, buscar un agente que ofrezca un determinado servicio y buscar todos los agentes que ofrezcan un determinado servicio. Estas funciones hacen que las interacciones entre los agentes y el *DF* sean transparente

para el usuario.

Una imagen ilustrativa de lo que puede ser una plataforma Magentix distribuída en dos host, la podemos ver en la Figura 4.

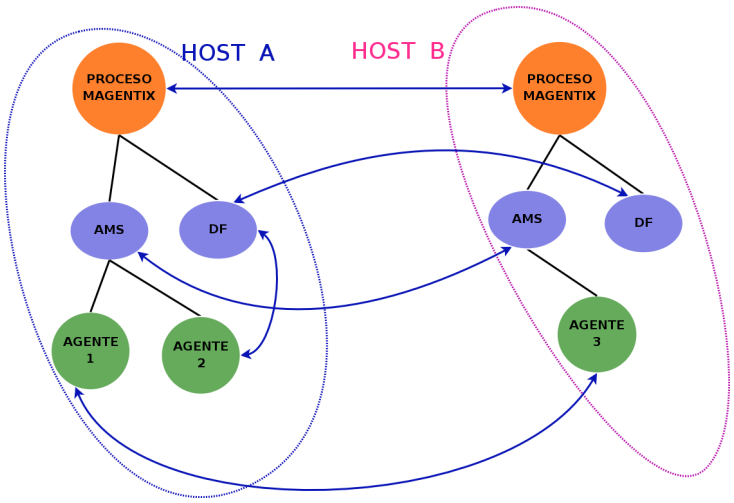

Figura 4: Arquitectura distribuída de la plataforma

El Servicio de Mensajería (MTS)

El servicio MTS se ha implementado como una biblioteca de funciones. El diseño del modelo de comunicación utilizado por los agentes tiene como objetivo favorecer la eficiencia y la escalabilidad de la plataforma, presentando una serie de características que lo hacen posible.

Los agentes se comunican siempre de forma P2P, sin utilizar ningún punto centralizador, con lo que se dota de una gran escalabilidad a las comunicaciones entre agentes. Para proporcionar la comunicación entre los distintos agentes de la plataforma, se ha implementado un servicio de mensajería entre agentes utilizando sockets TCP para ofrecer conexiones bidireccionales y en la medida de lo posible, permanentes. Cada agente MAGENTIX dispone de un socket servidor al que se podrán conectar otros agentes creando un socket cliente. Con todo esto, podemos decir, si lo miramos desde el punto de vista del paradigma cliente/servidor, que nuestros agentes son a la vez clientes y servidores de otros agentes.

El hecho de utilizar sockets TCP nos permite mantener una conexión abierta entre dos agentes. Está comprobado que mantener una conexión abierta entre dos procesos Linux para enviarse una gran cantidad de mensajes, es mucho más eficiente que abrir una conexión para enviar un único mensaje y volverla a cerrar [5]. Parece claro que si se aprovecha este hecho, dos agentes podrían tener abierta una conexión indefinidamente y enviarse mensajes cada vez que lo requiriesen consiguiendo un alto grado de eficiencia.

Lamentablemente, existe el inconveniente que un mismo proceso en Linux tiene un número limitado de sockets que puede tener abiertos, así cómo el número de descriptores que pueden estar abiertos en el SO es limitado. Puesto que un agente en Magentix queda implementado como un proceso de Linux, este inconveniente se translada a los agentes. Por otra parte, muchas de las interacciones entre cada par de agentes no implican un intercambio grande de mensajes, y por este sentido, puede que no sea necesario mantener todas las conexiones abiertas indefinidamente, porque muchas de ellas ya no se van a volver a utilizar.

Por este hecho se ha implementado a nivel de cada agente, una caché de conexiones abiertas con otros agentes (Figura 5). Cada agente dispone de una caché de conexiones TCP, con el fin de aprovechar al máximo las ganancias de mantener las conexiones TCP abiertas. Con esta caché se pretende mantener abiertas las conexiones correspondientes a las conversaciones más activas que tenga un agente, mediante una política LRU (Last Recent Used).

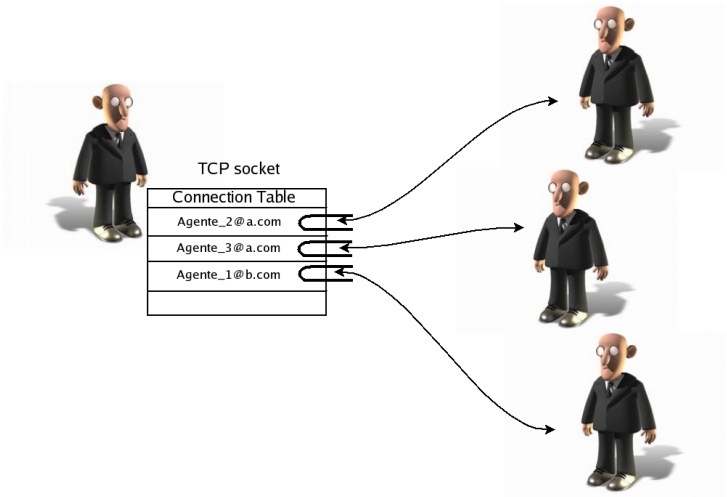

Figura 5: Modelo de comunicación

Cuando un agente quiere enviar un mensaje a otro, se comprueba si existe una conexión con el agente destinatario en la caché de conexiones. Si existe, se procederá al envío. Si no es así, se consulta la dirección del agente destinatario accediendo a la tabla GAT, y se abre una conexión TCP de forma que se establece un canal de comunicación bidireccional entre los dos agentes mediante el que cualquiera de ellos puede enviar un mensaje al otro. Por otra parte, en caso de haber llegado al máximo de conexiones permitidas para un agente, antes de crear una nueva se cerrará la que hace más tiempo que no se ha usado (LRU), y que se corresponde con una conversación con otro agente que lleva mucho tiempo inactiva (sin que ninguna de las partes se envíen un mensaje) y que probablemente vaya a permanecer más tiempo en ese estado. Con este diseño se llega a un compromiso entre la necesidad de mantener las conexiones abiertas el máximo tiempo posible y la imposibilidad de tener un número alto de conexiones abiertas durante un

periodo indefinido.

En este modelo cabe remarcar el hecho de que las direcciones físicas de los agentes (dirección IP del ordenador y puerto TCP) pueden ser consultadas de forma directa utilizando una tabla en memoria compartida GAT, con el fin de poder resolver la dirección de un agente a partir de su nombre de forma eficiente.

El hecho de utilizar sockets TCP a nivel de llamadas al sistema Linux, y memoria compartida para la consulta de las direcciones, dota al servicio de mensajería de una gran eficiencia. Además, la comunicación entre cualquier par de agentes de la plataforma es punto a punto, de manera que se obtiene un alto grado de escalabilidad.

Los Agentes en Magentix

Cada agente en Magentix es representado como un proceso independiente. Existe una biblioteca de funciones implementadas para ofrecer facilidad en la implementación de los agentes. Como se ha explicado en secciones anteriores. Los agentes tienen acceso de lectura a la tabla GAT. Este hecho permite ofrecer mayor flexibilidad a la hora de encontrar las direcciones de los distintos agentes con los que queremos comunicarnos. El hecho de que esta tabla se encuentre replicada en cada host por el servicio distribuído AMS, así como la tabla GST para el servicio *DF*, hace que cada agente, tenga la información más importante de cualquier otro agente de la plataforma (la dirección de contacto, los servicios ofrecidos, etc.) muy accesible.

Estructura de un agente

A nivel del SO, un agente es un proceso individual. Internamente, existe el hilo principal de ejecución correspondiente a todo proceso Linux, y además, dos hilos de ejecución adicionales para tratar los mensajes que recibe el agente (*receiver thread*) y para enviar los mensajes del agente hacia otro agente (*sender thread*).

El hilo principal de ejecución se encargará de realizar todas las gestiones iniciales del agente: inicializar el acceso a las tablas compartidas, inicializar las variables correspondientes, inicializar el módulo de comunicación y ejecutar el código propio del agente.

De entre estas funciones, la de inicializar el módulo de comunicación implica la creación de dos hilos especiales:*sender thread* y *receiver thread*. Estos hilos internamente serán los que gestionen los mensajes entrantes y salientes de las conexiones que tenga abiertas ese agente. Como hemos visto anteriormente, cada agente dispone de una tabla de conexiones que se corresponde con las conversaciones activas que mantiene. Estos hilos son los que se encargan de procesar los mensajes que se produzcan en estas conversaciones. El hilo de recepción recorrerá las conexiones abiertas para obtener mensajes que se hayan recibido por los sockets pertinentes. El hilo de envío de mensajes, cuando tenga que enviar algún mensaje, accederá a la tabla de conexiones y enviará por el socket correspondiente

del agente destino en cada momento. En caso de no tener una conexión abierta para ese agente destino, este hilo será el encargado de obtener la dirección física de contacto en la tabla GAT (dirección IP y puerto TCP asociados) y abrirá una nueva conexión, con lo que quedará en la tabla de conexiones. Por tanto, el acceso a la tabla GAT se realiza internamente por este hilo, quedando totalmente trasparente a nivel del usuario.

Otra de las funciones del hilo principal de este proceso es la de ejecutar el código propio del agente. Esto hace una llamada al método *mgx_ main* que haya implementado el usuario, utilizando los parámetros correspondientes.

Por ello, cada usuario que implemente un agente Magentix, deberá realizar un programa en C que contenga el método *mgx_ main*, incluyendo las librerías necesarias. En este método estará el código del agente que implemente el comportamiento del mismo, utilizando los parámetros deseados y utilizando el API ofrecido por las bibliotecas implementadas (Figura 6).

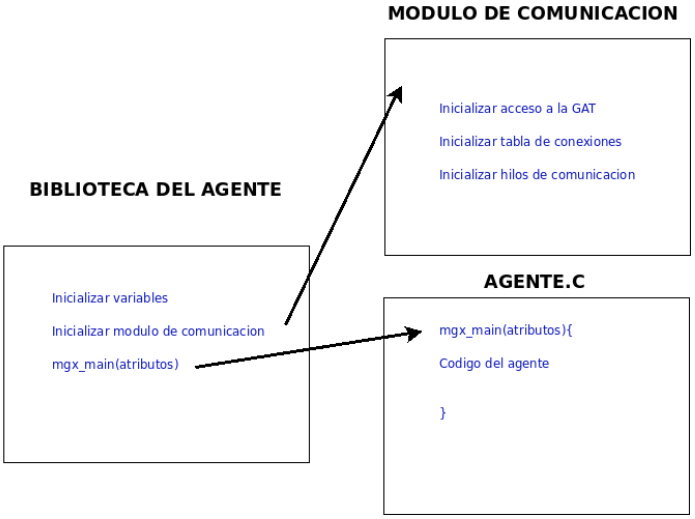

Figura 6: Inicialización de los módulos del agente

Mailboxes

En Magentix se introduce el concepto de *Mailbox* para ofrecer más versatilidad a la gestión de los mensajes de entrada. Un *Mailbox* funciona como un buzón de correos para distribuir los mensajes recibidos. Por defecto existe un único *Mailbox* por cada agente llamado *DEFAULT_ MAILBOX* donde se recibirán todos los mensajes destinados a ese agente.

En Magentix se ofrece la funcionalidad de crear nuevos *Mailboxes* y posteriormente asociarles identificadores de conversación. De este modo, cuando se reciba algún mensaje que en el campo *conversation_ id* figure dicho identificador de conversación, el mensaje será enrutado al *Maibox* correspondiente. Esta funcionalidad nos permitirá poder filtrar

y separar los mensajes que se reciben por este campo, pudiendo distribuir las diferentes conversaciones que tiene un agente entre los distintos *Mailboxes*. Un *Mailbox* no queda restringido a un único identificador de conversación, ya que se contempla poder asociar varios identificadores de conversación al mismo *Mailbox*.

La funcionalidad básica que debe tener un usuario de la plataforma en mente para trabajar con *Mailboxes* es la de crear nuevos *Mailboxes* y posteriormente asociar identificadores de conversación a estos *Mailboxes*.

Se define un API para la gestión de estos *Mailboxes*. El tipo definido para trabajar con *Mailboxes* es una estructura del tipo *mgx_mailbox_id_t* Las funciones de creación y eliminación de *Mailboxes* quedan definidas como:

```
int mgx_mailbox_create (mgx_mailbox_id_t * mailbox);
int mgx_mailbox_destroy (mgx_mailbox_id_t * mailbox);
```

Existen dos funciones especiales para asociar identificadores de conversación a *Mailboxes* ya creados y para eliminar estas relaciones:

```
int mgx_comm_set_route (char * conv_id, mgx_mailbox_id_t  mailbox);
int mgx_comm_delete_route (char * conv_id);
```

En la primera función se asocia un identificador de conversación a un *Mailbox*. Un mismo identificador de conversación no puede estar asociado a más de un *Mailbox*, pero un mismo *Mailbox* si que puede recibir mensajes con identificadores de conversación distintos. En la función *mgx_comm_set_route* asociará un identificador de conversación a un *Mailbox*, si este identificador de conversación ya estuviera asociado a otro *Mailbox*, se eliminará esta relación y se transvasarán los mensajes que hubiera en el antiguo *Mailbox* al nuevo. La segunda, permite desasociar esta relación entre un identificador de conversación y su *Mailbox* correspondiente.

Una imagen de la estructura interna del agente la podemos ver en la Figura 7

Estructura de los mensajes

Los mensajes que se intercambian los agentes Magentix se representan en RDF (Resource Description Framework) [3]. RDF se utiliza para identificar recursos web mediante URIs, para que la información pueda ser procesada por máquinas. Este lenguaje, utiliza el concepto de tripleta para representar la información. Cada tripleta está formada por un sujeto (el recurso al que hacemos referencia), un predicado (la propiedad asociada a dicho recurso) y un objeto (el valor de dicha propiedad) (figura 8).

Una descripción completa en RDF no sería un conjunto de tripletas. Los campos definidos en el estandard FIPA, están predefinidos como nodos RDF. (figura 9)

FIPA define la utilización de RDF como un posible lenguaje de contenido [1]. Utilizando esta especificación, se aumenta la capacidad de expresividad proporcionada. La

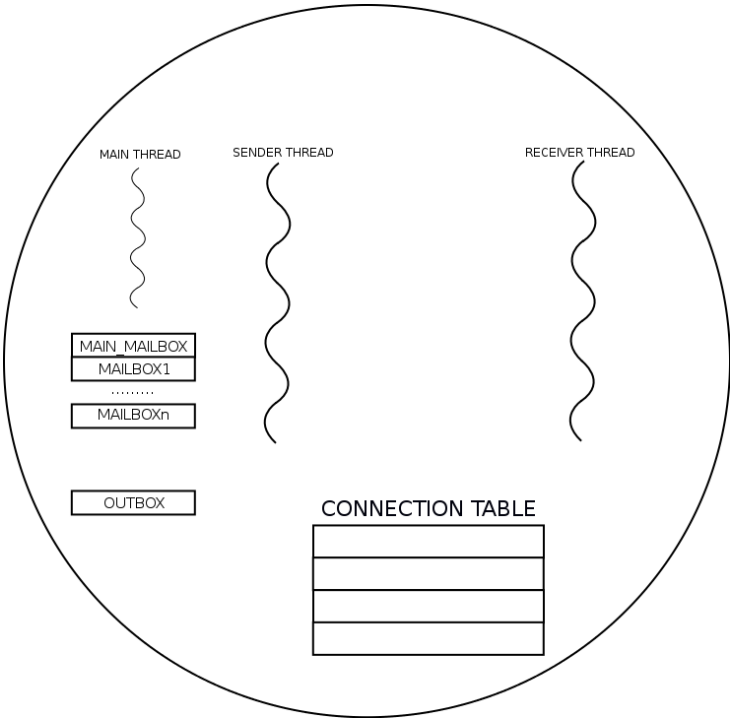

Figura 7: Estructura interna de un agente Magentix

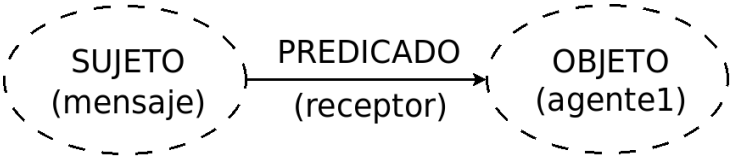

Figura 8: Tripleta RDF

mayoría de plataformas intercambian mensajes representados como cadenas de bytes. Esto reduce enormemente las posibilidades de representar la información, ya que, sería el propio usuario el que tendría que gestionar la manipulación de estas cadenas. Por contra, el hecho de utilizar tripletas para representar la información, nos añade una sobrecarga en la creación y la recuperación de todos los campos del mensaje.

Por defecto, en Magentix se ofrece una facilidad para gestionar los elementos definidos en el estandard FIPA [12]. El uso que se ofrece al usuario es la creación de tripletas en RDF, con los elementos de sujeto-predicado-objeto, a través de una biblioteca proporcionada. Para recuperar los elementos que aparecen en un mensaje, utilizaremos también la biblioteca, permitiendo obtener toda la información del mensaje, almacenándola por ejemplo, en variables.

En cuanto al API que se proporciona al usuario, existen dos funciones para enviar un mensaje:

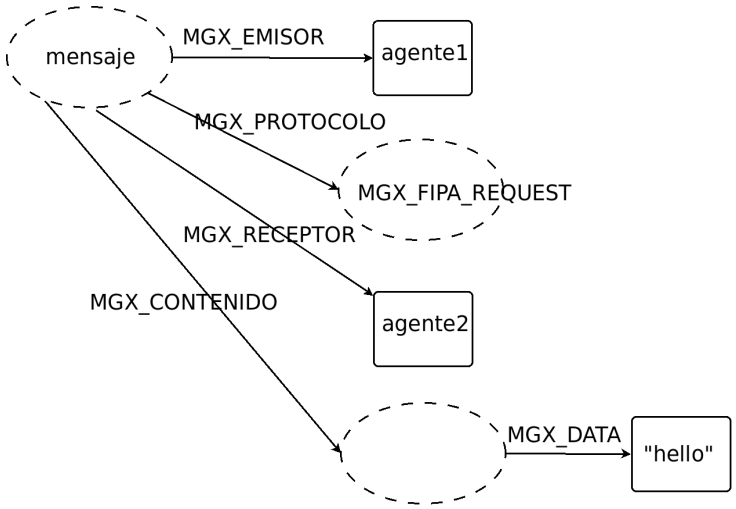

Figura 9: Estructura de un mensaje en Magentix

```
mgx_comm_send(mgx_rdf_model_t * message);
mgx_comm_send_and_destroy(mgx_rdf_model_t * message);
```

Las dos funciones envían el mensaje que se le pasa por parámetro. Como hemos explicado anteriormente, el mensaje se expresa en RDF y está representado en Magentix mediante un modelo RDF (*mgx_ rdf_ model_ t*). El destinatario del mensaje será un nombre de agente indicado en el campo *receiver* del mensaje. La diferencia entre ambas funciones es que la primera de ellas mantiene el mensaje que se le pasa por parámetro y la segunda libera la memoria que ocupa este mensaje. El uso de cada una de ellas depende de si se ha de reutilizar el mismo mensaje posteriormente o no.

Por otro lado, el API de Magentix ofrece tres funciones que se pueden utilizar cuando un agente de usuario ha de recibir algún mensaje:

```
mgx_comm_receive(mgx_mailbox_id_t mailbox, mgx_rdf_model_t *message);
mgx_comm_try_receive(mgx_mailbox_id_t mailbox, mgx_rdf_model_t *message);
mgx_comm_timed_receive(mgx_mailbox_id_t mailbox, mgx_r_model_t *message, struct
                  timespec timeout);
```

Las tres funciones recibirán un mensaje del Mailbox indicado como primer parámetro. La diferencia de las tres funciones radica en que la primera de ellas bloqueará al agente si no se encuentra ningún mensaje en el Mailbox indicado, y será desbloqueado cuando se reciba uno. La función *mgx_ try_ receive* intenta recibir un mensaje del Mailbox indicado, en caso que haya algún mensaje en ese buzón se recibirá, y si no lo hay, continuará la ejecución del agente. Finalmente, la función *mgx_ timed_ receive* suspenderá la ejecución del agente hasta que se reciba un mensaje en el Mailbox indicado, o hasta que venza el *timeout* pasado como tercer parámetro de la función en términos de una estructura de tipo *timespec*.

Unidades Organizativas en Magentix

Introducción

El concepto de organización de agentes nos ofrece un entorno que nos ayuda a simplificar, estructurar y desarrollar de una manera más flexible a la vez que potente, los SMA. Especialmente en SMA complejos, donde hay miles de agentes, es interesante el uso de este concepto para evitar el caos y lograr una mayor estructuración de la sociedad de agentes.

La propuesta principal de este trabajo es realizar un diseño para soportar no sólo interacciones entre agentes individuales sinó entre agentes y grupos de agentes. A partir de este soporte básico, podremos representar abstracciones de más alto nivel como las organizaciones de agentes.

En Magentix introducimos el concepto de unidades organizativas o simplemente unidades, como primer paso para dar soporte a las organizaciones de agentes. Una unidad es, conceptualmente, una abstracción que nos permite crear agrupaciones dinámicas o estáticas de agentes, ofreciendo una visión más potente a la hora de implementar SMA. Estas agrupaciones, nos permitirán crear multitud de entornos que con simples agentes serían imposibles o cuanto menos, más difíciles de implementar y controlar. El hecho de utilizar el concepto de unidad organizativa hará más sencillo el diseño de SMA complejos así como la coordinación de los elementos que lo forman.

A nivel de la plataforma, una unidad será una agrupación de agentes con ciertas propiedades, pero a un nivel más alto, una unidad o un conjunto de unidades podrían ser la base para componer lo que llamamos una organización de agentes. Para cada organización de agentes podremos definir un conjunto de miembros que coordinan sus actividades para conseguir unos ciertos objetivos, llevando a cabo planes o estrategias y respetando unas normas, prohibiciones y obligaciones asociadas a su condición como elementos del grupo.

Tal y como se define en [11], la coordinación entre los distintos miembros de la organización es fundamental para llevar a cabo los objetivos de una manera coherente. La manera en que coordinan las actividades los miembros de una organización dependerá de la arquitectura organizacional de la misma, defiendo una serie de paradigmas de organizaciones que agrupan los distintos agentes, estos paradigmas pueden ser jerarquías, federaciones, coaliciones, grupos, etc.

Para dar un esquema similar al anterior, se requiere partir de una base que permita la implementación a un nivel más alto. Para ello, se definirá la posibilidad de crear unidades organizativas.

Por otro lado, en el análisis de los SMA, hay propuestas metodológicas que definen un modelo de roles. En estos modelos se identifican los roles que están presentes en el sistema, de manera que los agentes que forman una organización interaccionan y se coordinan entre ellos en función de los roles que jueguen. Los roles tendrán asociados unas tareas y habilidades, definiendo claramente los comportamientos de los agentes dentro de la organización en función de los protocolos de interacción entre los distintos roles. Además, estos roles se rigen mediante unas normas de comportamiento, las cuales definen qué tiene permitido o no hacer cada agente dentro de la organización, y que está obligado a hacer en función de las interacciones entre los distintos agentes.

Con estas premisas, definiremos como unidad organizativa una agrupación de agentes, que juegan unos roles definidos y que se coordinan entre si para llevar a cabo unos objetivos.

Dependiendo del rol que desempeñe cada miembro de una organización, se permitirán unas interacciones entre ellos la unidad organizativa definirá además las interacciones posibles entre los distintos roles (figura 10). Como podemos ver, en esta unidad hay un tipo de agente que se comunica con todos los demás, y todos ellos sólo tienen relación con éste. Representaría un ejemplo de arquitectura jerárquica, dónde existe un agente que juega el rol de supervisor y una serie de agentes que son los subordinados. La relación entre los agentes será de supervisor a subordinado y nunca entre dos subordinados entre si.

Definiendo los distintos roles de una unidad así como una serie de normas sujetas a estos roles, podemos crear relaciones entre las distintas unidades organizativas, para construir así unidades más complejas que representen arquitecturas de burocracias, federaciones... (figura 12). Estas unidades más complejas, si mantienen un objetivo global sujeto a unas normas, las podremos considerar en un futuro como organizaciones de agentes.

Un primer paso para poder ofrecer organizaciones en Magentix, es ampliar las posibilidades de interacción entre los elementos de la plataforma, es decir, permitir la interacción entre agentes y unidades organizativas.

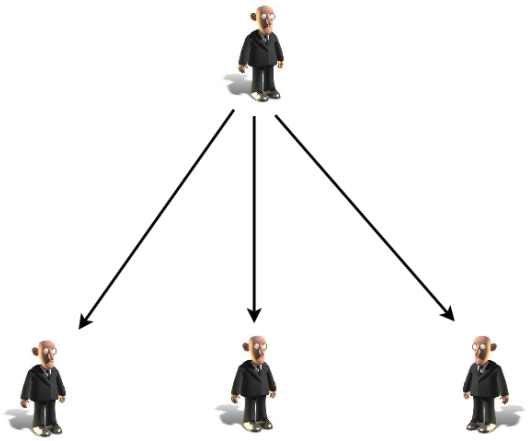

Figura 10: Relaciones entre los roles de una unidad organizativa

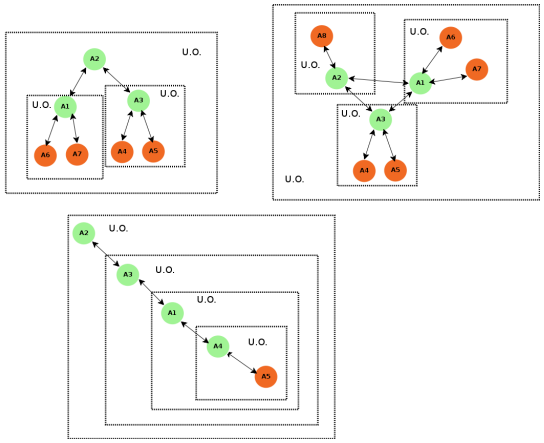

Figura 11: Relaciones entre las distintas unidades

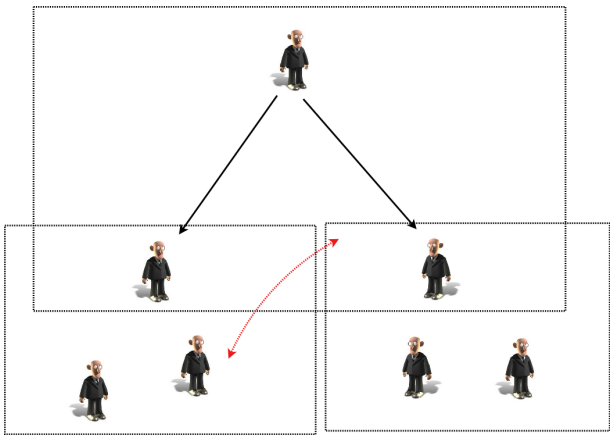

Figura 12: *UO interrelacionadas*

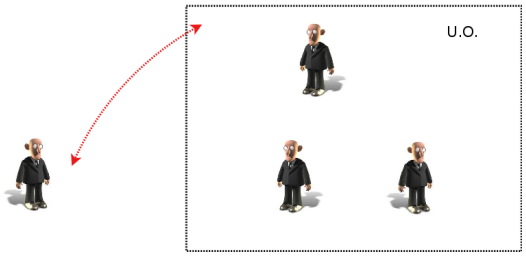

Figura 13: *Relación entre agente y unidad*

En las figuras 12 y 13 podemos ver como algunos miembros de unas unidades interactúan con otras unidades y no con elementos individuales. Para ello definimos una comunicación posible con unidades organizativas.

Realmente, la comunicación con una unidad organizativa debería ser transparente de cara al agente. De manera que la unidad debe ser como una caja negra que realiza unas transacciones internas para obtener un resultado.

En un ejemplo simple (figura 14) podemos ver una posible transacción. Tenemos una unidad organizativa que ofrece una funcionalidad de distintos servicios registrados (cines, museos, teatros...). Los agentes, pueden interactuar con esta unidad para obtener la información requerida. El agente *demandante* quiere obtener cierta información relacionada con los servicios de cine. Tal y como se ha definido la arquitectura topológica de dicha unidad, los agentes no interaccionan directamente con el agente que juega el rol de *cine*, sino que se define un *broker* el cuál contacta con un agente de *espectáculos* y éste, a su vez, con el agente de *cine*. Pero todo esto es transparente al agente *demandante*, ya que la unidad es como una caja negra. De esta forma, el agente oferente se comunica con la unidad de *servicios* y recibe la respuesta de la unidad de *servicios*.

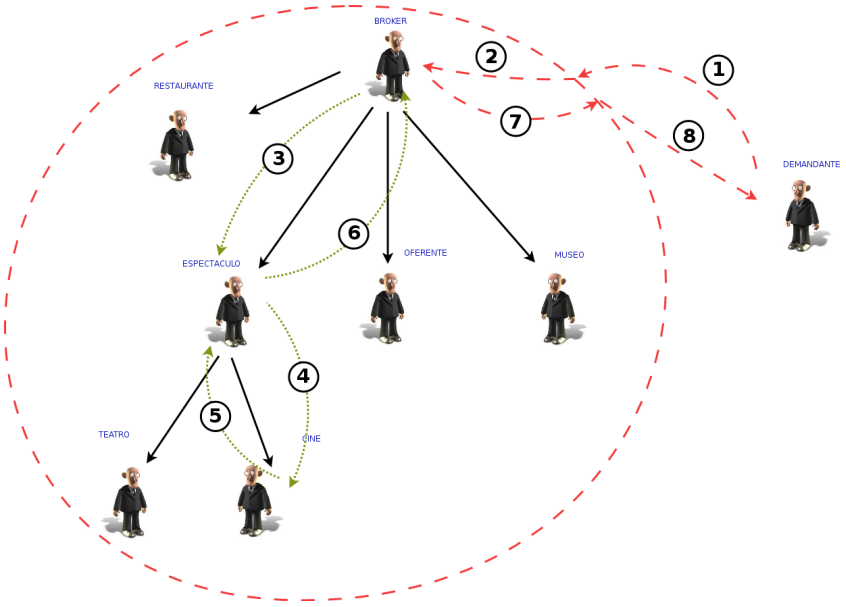

Figura 14: Unidad de servicios

Este ejemplo muestra una posible organización típica de agentes, dónde los mensajes que se reciben del exterior, en este caso son tratados por el agente *broker*. Ya tenemos dos requisitos que debemos tener en cuenta: una unidad organizativa estará formada por un grupo de agentes, y deberemos definir unos agentes que serán los encargados de recibir los mensajes del exterior. Ello nos permitirá modelar las estructuras organizativas típicas en diseños a más alto nivel. Estos agentes los definiremos como los agentes de contacto asociados a la unidad, aunque para los agentes externos esto es transparente, ya que en

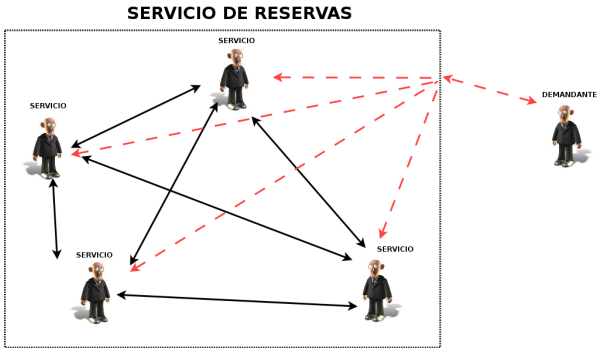

Figura 15: Servicio de reservas

este caso, el agente *demandante* desconoce que sus mensajes van destinados a un agente *broker*.

La posibilidad de definir unidades organizativas, y agentes de contacto asociadas a ellas, nos abre las puertas a una gran cantidad de posibles combinaciones para formar relaciones entre agentes y unidades.

Aparte de las organizaciones típicas de agentes, podemos utilizar el concepto de unidad organizativa para otros fines, como por ejemplo agrupar un conjunto de agentes con un servicio idéntico. En la figura 15 se muestra un servicio de reserva de billetes. Todos los agentes que forman esta unidad, ofrecen el mismo servicio, pero están agrupados en la unidad. De esta manera, cuando un agente externo requiera utilizar el servicio de reservas, no contactará con cualquier agente que ofrezca el servicio, sinó con la unidad (aunque repetimos, para el agente externo la interacción es como si la realizara con un agente individual). En este caso, la petición recibida por la unidad, será reenviada a todos los agentes internos de la misma, quiénes se coordinarán entre ellos para atenderla, según las políticas que tengamos definidas. En este esquema, varios agentes de la unidad son los agentes de contacto definidos para esa unidad, es decir, los agentes que tratarán dicha petición.

También podemos tener un esquema similar, pero que no requiera que todos los mensajes sean atendidos por todos los agentes. En escenarios dispersos, dónde se requiera una cierta distribución de la carga, podemos distribuir los mensajes que se dirigen a la unidad, entre los distintos agentes internos, de una manera alternada (figura 16). El criterio para decidir qué agente de la unidad atenderá cada petición puede ser completamente aleatorio, distribuído mediante alguna política circular que reparta las peticiones, o simplemente atendiendo a cuestiones de la tipología de la red, es decir, dividir las peticiones recibidas según la procedencia de la misma (figura 17).

La posibilidad de definir unidades nos permite cambiar dinámicamente qué agente atenderá las peticiones de dicho servicio. Otro escenario dónde podemos utilizar este concepto sería para "encapsular" una funcionalidad. En la figura 18 podemos ver que tenemos un único agente que ofrece un servicio de reservas, pero este servicio no es ofrecido

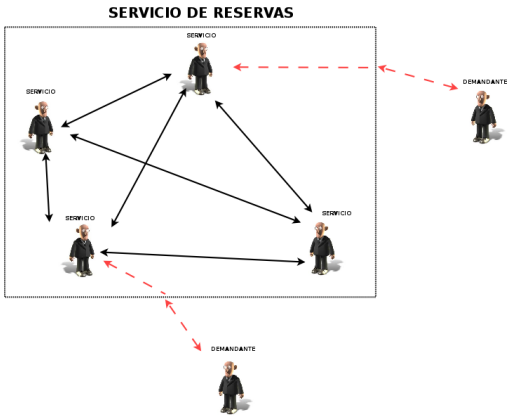

Figura 16: Servicio de reservas

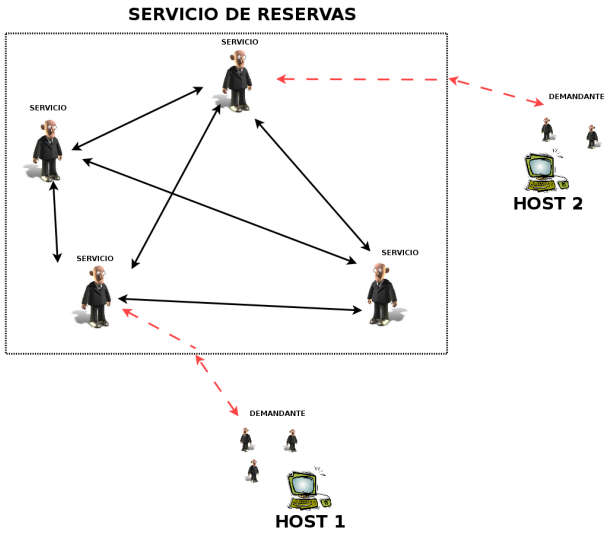

Figura 17: Servicio de reservas

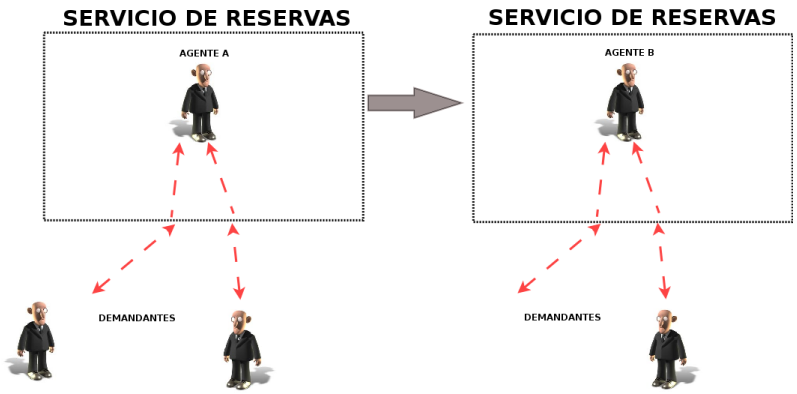

Figura 18: Servicio de reservas

por el agente individual sinó que es ofrecido a través de una unidad. De esta manera, supongamos que decidamos cambiar el agente interno por otra versión más actual, o simplemente se haya caído ese agente y necesitemos reemplazarlo "en caliente". Con este enfoque, se puede realizar el cambio sin que esto afecte a los agentes externos, ya que hemos enmascarado el servicio a través de la unidad (figura 18).

De esta manera, planteamos la comunicación con unidades organizativas, definiendo uno o varios agentes que tratarán las peticiones a dichas unidades según unos criterios o políticas de enrutamiento. Este diseño será la base para ofrecer en un futuro el soporte de organizaciones, pero además, esta versatilidad de la comunicación, nos ofrece más posibilidades de interactuación, independientemente de que las agrupaciones de agentes sean organizaciones, como ya hemos visto.

Antes de llegar a este nivel de complejidad hay que modificar la arquitectura interna de Magentix para dar cabida al concepto de unidad organizativa y al concepto de comunicación con unidades organizativas.

Implementación en Magentix

En esta sección plantearemos cómo integrar los conceptos básicos vistos en el punto anterior en la plataforma Magentix.

Necesitamos ofrecer la posibilidad de agrupar agentes. De manera que tengamos unidades organizativas y miembros de estas unidades. Estos miembros formarán parte de la unidad jugando un cierto rol, que en un futuro, será lo que interrelacione los distintos miembros de la unidad, definiendo los protocolos de interacción entre los roles y las tareas y estrategias asociadas a cada rol.

Como hemos visto anteriormente, necesitamos incorporar distintas alternativas para enviar los mensajes a las unidades. Las unidades organizativas serán como unas cajas negras de cara a los agentes externos, y éstos se comunicarán con ellas. Pero esta comuni-

cación debe ser enrutada por la plataforma. Para ello necesitamos cambiar la manera en la que se entregan los mensajes en Magentix. En las siguientes secciones veremos cómo podemos definir las diferentes alternativas para la comunicación con unidades organizativas.

El servicio OUM (Organizational Unit Manager)

Implementaremos la gestión de las distintas unidades organizativas mediante un nuevo servicio de la plataforma llamado *servicio de gestión de unidades organizativas* (*OUM*). Este servicio estará distribuído en todos los hosts de la plataforma y almacenará toda la información necesaria de cada unidad que se encuentre en Magentix. Esta información será gestionada internamente por Magentix, aunque en un futuro pueda servir a los agentes externos para obtener más información de las organizaciones, como los objetivos de la misma, la estructura topológica, los miembros, etc.

El servicio *OUM* será también la entidad con la que interaccionarán los agentes para poder crear nuevas unidades, así como añadir miembros o cambiar sus opciones. Por tanto, al igual que el *DF*, el servicio *OUM*, tendrá una interfaz similar a la de los agentes que le permitirá interactuar con ellos mediante el paso de mensajes.

La implementación de dicho servicio es similar a la realizada para los otros servicios de la plataforma. El servicio *OUM* se compone de una serie de procesos llamados *oum* y que se lanzan en cada host de la plataforma. Cada proceso *oum* está creado y controlado por el proceso *magentix* de ese mismo host. Cada proceso *oum* guarda la información asociada a todas las unidades que se encuentran en la plataforma utilizando una tabla llamada **GUT** (Global Unit Table).

La tabla GUT (Global Unit Table)

La tabla **GUT** es una tabla replicada en cada proceso *oum* de la plataforma y que contiene la información referente a cada unidad de la plataforma y que será requerida por magentix para la gestión de las mismas.

Esta tabla se implementa como una tabla hash indexada por el nombre de las unidades, de manera que la manipulación de la misma tiene unos costes promedios constantes. La información contenida en dicha tabla será de lectura y escritura para los procesos *oum* y de lectura para los agentes, con el fin que puedan comunicarse eficientemente con las unidades.

Los atributos de cada unidad organizativa almacenados en la tabla *GUT* son:

- **nombre**(name): cada unidad tiene que tener un nombre único dentro de la plataforma. Este nombre está representado como una cadena de caracteres y debe identificar unequívocamente a cada unidad de la plataforma. A fin de que los agentes realicen una comunicación transparente independientemente de si interaccionan con una uni-

dad o un agente individual, el nombre de la unidad tiene que ser también diferente de cualquier agente de la plataforma.

- **agentes de contacto**(contact agents): tal y como hemos visto en el apartado anterior, cada unidad tiene asociado uno o más agentes que serán los que recibirán los mensajes destinados a la misma. Con el fin de ofrecer la máxima flexibilidad posible a la hora de implementar distintos tipos de unidades, se definirán también, distintas políticas para la entrega de los mensajes destinados a la unidad. Dependiendo de estas estrategias, cada unidad tendrá asociado uno o más agentes de contacto. En caso de haber más de un agente de contacto estarán representados mediante una lista enlazada. Para cada agente de contacto se guarda la siguiente información:

  - *nombre*(name): es el nombre del agente a nivel de la plataforma. Es decir, es el nombre con el que queda registrado el agente en la GAT.
  - *host*: es la dirección IP asociada al agente almacenada en la GAT.
  - *puerto*: es el puerto asociado al agente que está almacenado en la GAT.

  Cuando se define un agente como agente de contacto, el servicio *OUM* contacta con el *AMS* y a partir del nombre del agente (name) recupera la IP y el puerto.

- **tipo de enrutamiento**(routing type): relacionado con el campo anterior, definimos distintas estrategias para la recepción de los mensajes de la unidad. Cuando un agente interacciona con una unidad organizativa, en última instancia el mensaje será entregado a uno o más agentes de contacto. La unidad es un concepto abstracto y por tanto, por si misma no tiene la capacidad de recibir mensajes. La manera de definir la recepción de los mensajes destinados a la unidad será mediante la definición de uno o más agentes que se encarguen de dicha tarea. Paralelamente, se define una serie de tipos de enrutamiento que definirán a qué agente se le entrega el mensaje. Estos tipos han sido seleccionados atendiendo a las distintas posibilidades de unidades organizativas analizadas en el punto .
  En un primer lugar se plantea la posibilidad de asocir uno o varios agentes de contacto a la unidad, para ello se contemplan los tipos de enrutamiento *UNICAST* y *MULTICAST*.

  - *UNICAST*: si la unidad tiene asociada este tipo de enrutamiento, tendrá definido uno y solo un agente que se encargue de recibir los mensajes. Todo mensaje destinado a la unidad será recibido por este agente, quién ya interpretará dichos mensajes según defina el usuario. Este tipo nos permitirá en un futuro, definir distinas estructuras topológicas (por ejemplo una jerarquía de agentes).
  - *MULTICAST*: definiendo este tipo de enrutamiento podemos tener asociados más de un agente para recibir los mensajes. Todo mensaje destinado a la unidad se enviará a cada uno de los receptores, por tanto, todos estos agentes tendrán la capacidad de tratar los mensajes recibidos por la unidad, según defina el usuario.
    También se plantean distintos tipos donde el agente de contacto asociado a la

unidad no es importante y lo que se busca es una distribución de la carga. Para ello se plantean tres tipos más de enrutamiento: *ROUND ROBIN*, *RANDOM* y *SOURCEHASH*.

- *ROUND ROBIN*: definiendo este tipo de enrutamiento podemos tener asociados varios agentes de contacto para la unidad. Sin embargo, cada mensaje destinado a dicha unidad será entregado a un único agente siguiendo una política de round robin. De esta manera, ningún agente de contacto actuará como cuello de botella ya que los mensajes destinados a la unidad serán repartidos uniformemente entre los distintos agentes de contacto.
- *RANDOM*: esta política es similar a la anterior salvo que el criterio de distribución de la carga tiene un caracter aleatorio. Con este tipo podemos tener varios agentes asociados, pero sólo uno recibirá el mensaje. Además, el receptor que lo reciba no debe ser importante para el usuario ya que se entregará a uno de los agentes de contacto elegido al azar.
- *SOURCEHASH*: con este tipo de enrutamiento se pretende conseguir equilibrar la carga de la plataforma repartiendo las peticiones recibidas entre los distintos agentes de contacto, dependiendo de la procedencia de dichas peticiones. Se pretende por tanto, distribuir los mensajes enviados a la unidad a los distintos agentes de contacto en función de su ubicación física desde el punto de vista de la tipología de la red. Por el momento, sólo se contempla si el agente emisor se encuentra en el mismo host que algun agente de contacto asociado a la unidad. Es decir, entre dos agentes que estén asociados a la unidad, recibirá el mensaje aquel que se encuentre en el mismo host que el agente que envía el mensaje. Esta implementación aún no está integrada en la plataforma.

- **miembros**(members): como hemos analizado en la sección anterior (sección), necesitaremos tener la información de los agentes que forman parte de la unidad. Por ello, todos los agentes que forman parte de la unidad organizativa están almacenados en este campo. Los miembros se representan mediante una lista enlazada y para cada miembro tenemos la siguiente información:

- *nombre*(name): es el nombre del agente a nivel de la plataforma. Es decir, es el nombre con el que queda registrado el agente en la GAT.

- *rol*: es un rol que tiene asociado ese agente dentro de la unidad. Se representa mediante una cadena de caracteres. Puesto que un agente puede tener más de un rol dentro de la misma unidad, esto se representará mediante varios miembros donde el campo *nombre* es el mismo y lo que cambia es el *rol*.

- **manager**: toda unidad organizativa tiene un manager asociado. El manager por defecto es el agente que crea la unidad y inicialmente será el único de la unidad capaz de insertar o borrar nuevos miembros, modificar los agentes de contacto o eliminar la unidad. El manager está representado de la misma manera que un agente de contacto (nombre, host y puerto) y por defecto, el primer agente de contacto asociado a la unidad será el mismo manager.

Manteniendo toda esta información, Magentix ya podrá enrutar los distintos mensajes asociados a cada unidad. Además de disponer de la lista de miembros de la unidad que nos servirá en un futuro para definir las relaciones entre ellos en función de los roles que jueguen.

Esta tabla estará almacenada y replicada en cada uno de los hosts. Puesto que en cada host hay un proceso *oum*, cada vez que uno detecte un cambio en alguna unidad deberá transmitir la información al resto de los *oum* de la plataforma para mantener actualizadas todas las tablas. La forma de actualizar esta información es idéntica a la utilizada por los otros servicios, es decir, mediante la utilización de comando entre los procesos *oum* de la plataforma. El *oum* que detecta el cambio envía un comando al *oum* situado en el host principal y éste ya se encarga de enviar un comando al resto de *oum* de los otros hosts (figura 19).

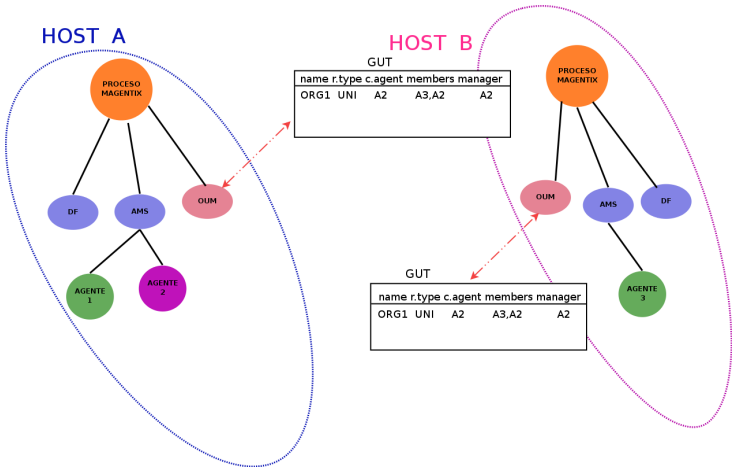

Figura 19: Arquitectura distribuída del servicio *OUM*

Los cambios que puede detectar un *oum* son los siguientes:

- Se ha creado una nueva unidad
- Se ha eliminado una unidad existente
- Se ha insertado un nuevo agente de contacto
- Se ha eliminado un agente de contacto
- Se ha insertado un nuevo miembro
- Se ha eliminado un miembro
- Se ha modificado el manager de la unidad

Además, cada vez que se une un host a la plataforma, toda la información de la GUT se tiene que replicar en la GUT del nuevo proceso *oum*. Para agilizar el proceso también se permiten los siguientes comandos.

- Insertar un listado de agentes de contacto
- Insertar un listado de miembros

Interacción con el servicio OUM

Los distintos procesos *oum*, además de comunicarse entre sí mediante comandos, tienen una interfaz para comunicarse con los agentes de la plataforma. De esta manera, un agente puede interaccionar con el servicio *OUM* (con el proceso *oum* de su mismo host) mediante el envío de mensajes.

Un agente puede comunicarse con el servicio *OUM* para crear nuevas unidades, o modificar la información de una unidad existente siempre y cuando sea el manager de la misma. Esto se realiza mediante el envío de mensajes destinados al *OUM_SERVICE* que se enviarán al puerto bien conocido del proceso *oum* de ese host. A fin de hacer más cómodo este proceso para el usuario, se encapsulan estos mensajes mediante una API básica para gestionar este tipo de unidades:

```
int mgx_agent_new_unit(char *unit_name, int routing_type);
int mgx_agent_del_unit(char *unit_name);
int mgx_agent_new_member(char *unit_name, char *agent_name, char *rol_name);
int mgx_agent_del_member(char *unit_name, char *agent_name, char *rol_name);
int mgx_agent_new_contact_agent(char *unit_name, char *agent_name);
int mgx_agent_del_contact_agent(char *unit_name, char *agent_name);
int mgx_agent_new_manager(char *unit_name, char *agent_name);
```

Hay que observar que el agente no requiere proporcionar más información en estas llamadas. El resto de campos que necesita el *OUM* guardar en la *GUT* (IP's y puertos por ejemplo) los obtiene, mediante comandos con el *AMS*.

La especificación más detallada de estás funciones se puede ver en el manual de usuario .

Tal vez en el ejemplo del *DF* era menos intuitivo, pero en este caso se pueden observar los beneficios nos aporta el uso de un lenguaje de contenido para la transmisión de la información, en nuestro caso, RDF. Supongamos que queremos insertar una unidad de nombre *unit1*. Esta unidad tendrá 3 miembros *agent1*, *agent2* y *agent3*. Como agentes de contacto estará el *agent1*, que además será el manager de la misma, y por lo tanto, el tipo de enrutamiento será *unicast*. Esta información, transmitida como tripletas es fácil de transmitir y de recuperar para el que lo reciba. Ya que el agente receptor del mensaje (o en este caso, el propio *oum*, recuperará el objeto de una tripleta (*agent1*), conociendo el

sujeto (*unit1*) y el predicado (*manager*). Si toda esta información tiene que transmitirse, como en muchas de las plataformas existentes, como cadenas de caracteres, el usuario sería el encargado de parsear todos los contenidos de los mensajes, con los posibles fallos que haya de parseo (ya que resulta más complicado comprobar la correctitud de la información que se envía) y la dificultad de ampliar la información que puede recuperar, ya que siempre se trataría de un proceso hecho ad-hoc a los requisitos.

Enrutamiento de los mensajes

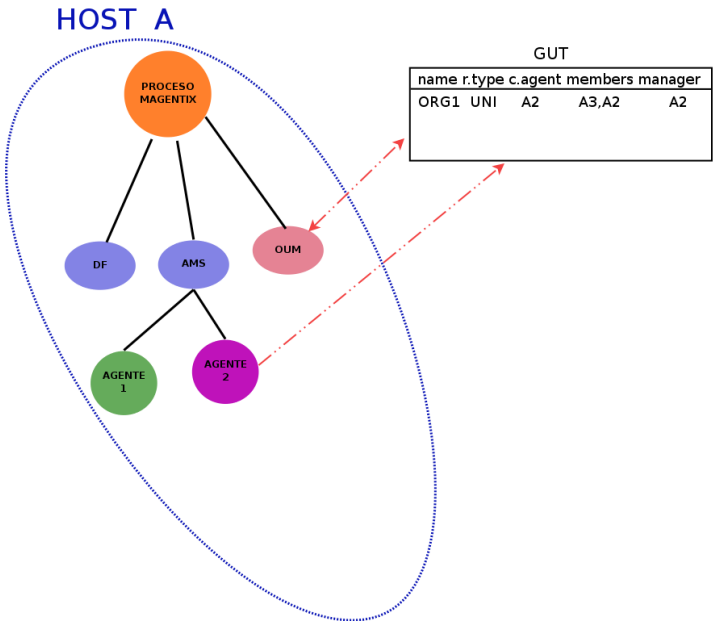

Figura 20: Consultas a la GUT

Cuando un agente envía un mensaje a una unidad, realmente no sabe si se trata de una unidad o de un agente individual ya que utiliza una cadena de caracteres que representa el nombre. Por ello, se ha modificado el módulo de comunicación de la plataforma.

Recordemos que cada vez que un agente envía un mensaje se consulta el agente receptor en una caché de conexiones y si no se encuentra se busca su dirección IP y su puerto en la tabla GAT, que recordemos que está mapeada en memoria compartida para que la puedan leer los agentes. Teniendo en cuenta que podemos disponer de unidades en la plataforma, se requiere modificar este modelo para incluir la gestión de la GUT.

Cuando el destinatario del mensaje no se encuentra ni en la caché de conexiones ni en la tabla GAT, se realiza una búsqueda en la tabla GUT para ver si se trata de una unidad. En caso afirmativo se tiene que recuperar el destinatario del mensaje, y esto dependerá de la política de enrutamiento definida para esa unidad (figura 20).

Si la unidad tiene definida un tipo *UNICAST*, *RANDOM*, *SOURCEHASH* o *ROUN-*

DROBIN se recupera ese agente y se envía como si de un agente individual se tratara, guardando posteriormente la información en la caché. En cambio, si el tipo es *MULTICAST* no se puede guardar ninguna dirección en la caché de conexiones, ya que el mensaje se tiene que enviar a todos los agentes asociados.

Manual de Usuario para Unidades

In Magentix is defined the concept of organizational unit, or just unit. A unit is an entity for grouping some agents. Agents wich offers simple services can be grouped to offer more complex services. Agents can be grouped building a topological structure to solve some tasks, share out information and cooperate among them. Or Agents that just offer similar services could be grouped to coordinate and distributed their requests, and so on.

A unit in magentix is shown like a simple agents. Thus, some agent wich requieres contact to a unit, can send messages to it, in a transparent way without knowing wich agents compose the unit and even knowing that the receiver of its message is a unit.

Magentix platform implements a mechanism for enrouting messages to the units by means of assigned agents. Each unit has an identifier (agent name) like any agent of the platform. Thus, when an agent sends a message, it doesn't know if receiver is a single agent or a unit. When we design a unit, we must specify at least one agent wich is in charge of receiving the messages addressed to the unit. Thus, when some agent sends a message to a receiver by means of a logical name, if this name is associated to a unit, Magentix platform will route the message to the corresponding agent. The agents wich receives messages addressed to the unit are labeled as *contact agents*.

The agents wich compose the unit are labeled as *members*. Each member in Magentix plays a *rol* inside the unit. That can help users to define topological structures for representing their organizational units.

Moreover, each unit has associated a *manager*. Only the manager of the unit can add or delete new members, contact agents, and is the single agent of the unit wich can delete the unit. The manager can be changed but by default is the agent that creates the unit.

For routing messages addressed to units, we use the concept of *routing type*. When we create a new unit, we must specify wich kind of routing will be used to the unit.

According to this parameter, the way of delivering the messages to the contact agents will be different.

For managing this concept a new service is integrated in Magentix: the Organizational Unit Manager (OUM). This service like other services of the platform is distributed among the different hosts. Moreover, agents can send messages to this service for retrieving information of any unit and for the management of the units. We can send messages addressed to this service adding the *OUM_SERVICE* receiver in the receiver field of the message.

For managing the creation, deletion and modify of units, we have dessigned a basic programming interface easy to use which encapsulates this messages traffic.

Defines

- #define **OUM_UNIT_ROUTING_TYPE_UNICAST** 1
- #define **OUM_UNIT_ROUTING_TYPE_MULTICAST** 2
- #define **OUM_UNIT_ROUTING_TYPE_ROUNDROBIN** 3
- #define **OUM_UNIT_ROUTING_TYPE_SOURCEHASH** 4[1]
- #define **OUM_UNIT_ROUTING_TYPE_RANDOM** 5

*****

Functions

- int **mgx_agent_new_unit** (char *unit_name, int routing_type)

  Function for creating of a new unit. Starting point.

- int **mgx_agent_del_unit** (char *unit_name)

  Function for deleting a unit.

- int **mgx_agent_new_member** (char *unit_name, char *agent_name, char *rol_name)

  Function for adding a member to the unit.

- int **mgx_agent_del_member** (char *unit_name, char *agent_name, char *rol_name)

  Function for deleting a member from the unit.

- int **mgx_agent_new_contact_agent** (char *unit_name, char *agent_name)

[1] This option is not implemented yet

Function for adding new agents for representing the unit.

- int **mgx_agent_del_contact_agent** (char *unit_name, char *agent_name)

  Function for deleting a contact agent from the unit.

- int **mgx_agent_new_manager** (char *unit_name, char *agent_name)

  Function for updating the manager of the unit.

Define Documentation

#define OUM_UNIT_ROUTING_TYPE_UNICAST 1

A single agent is the receiver of the messages addressed to the unit. By default every message sended to the unit will be received by the manager of the unit. But this receiver can be changed by calling the function mgx_agent_del_contact_agent.

#define OUM_UNIT_ROUTING_TYPE_MULTICAST 2

Some agents are associated as receivers of the messages sended to the unit. When an agent sends a message to a unit that has been created using this routing type, the message is sended to every contact_agent defined in the unit.

#define OUM_UNIT_ROUTING_TYPE_ROUNDROBIN 3

In this kind of routing type the user can define some contact agents to the unit. But every sended message is delivered to a single one using a Round Robin policy. For example, if a unit has associated two contact agents, every message addressed to the unit will be sended to the first contact or the second one using a round robin policy. Thus, every message is addressed to a single agent.

#define OUM_UNIT_ROUTING_TYPE_RANDOM 5

Defining this routing type it doesn't matter the contact agent who receives the messages. Using a random function every message is delivered to one of the contact agents defined in the unit. Absolutely, if there are a single agent defined as a contact agent, every message will be delivered to it.

Function Documentation

int mgx_agent_del_contact_agent (char * *unit_name*, char * *agent_name*)

Function for deleting a contact agent from the unit.

This function allows user deleting a contact agent from the unit. When an agent is deleted as contact agent, it can't receive messages addressed to the unit. It is important to note that every unit must have at least one contact agent for receive the messages. By default the manager of the unit is the contact agent defined.

Parameters:

← ***unit_name*** The name of the unit. If the unit does not exist, this function has no effect.

← ***agent_name*** The name of the agent. If the agent is no a contact agent of the unit, this function has no effect.

int mgx_agent_del_member (char * *unit_name*, char * *agent_name*, char * *rol_name*)

Function for deleting a member from the unit.

This function allows user to delete some agent playing an specific rol from the unit. Thus, the agent can still be member but playing different rols.

Parameters:

← ***unit_name*** The name of the unit. If the unit does not exist, this function has no effect.

← ***agent_name*** The name of the agent. Agent must be member of the unit. If not, this function has no effect.

← ***rol_name*** The name of the rol associated to the agent. If the agent specifyed is no playing the rol, this function has no effect.

int mgx_agent_del_unit (char * *unit_name*)

Function for deleting a unit.

This function allows the user to delete a unit. Only the manager of the unit can delete the unit. When this function is called, the unit is deleted from the GUT and every message addressed to the unit will be ruled out by the platform.

Parameters:

← ***unit_name*** The name of the new unit. If the unit does not exist, this function has no effect.

int mgx_agent_new_contact_agent (char * *unit_name*, char * *agent_name*)

Function for adding new agents for representing the unit.

This function allows user adding new agents to be contact agents of the unit. That is, they can receive some messages addressed to the unit depending on the routing type defined on the method mgx_agent_new_unit.

If the routing type is OUM_UNIT_ROUTING_TYPE_UNICAST, this function updates the previous contact agent by the new contact agent. If another type is defined for the unit, this new contact agent will receive messages according the specific policy: OUM_UNIT_ROUTING_TYPE_MULTICAST, OUM_UNIT_ROUTING_TYPE_RANDOM, OUM_UNIT_ROUTING_TYPE_SOURCEHASH, OUM_UNIT_ROUTING_TYPE_ROUND_ROBIN.

Parameters:

← ***unit_name*** The name of the unit. If the unit does not exist, this function has no effect.

← ***agent_name*** The name of the agent. Agent must be registered in the GAT.

Internally, OUM service contacts with AMS service to obtain the contact information: IP, and port of the agent.

int mgx_agent_new_manager (char * *unit_name*, char * *agent_name*)

Function for updating the manager of the unit.

This function allows user modifying the manager of the unit. As we previously said, only a single manager is allowed per unit. Thus, when calling this function the manager is changed. In spite of changing the manager, if the previous was a contact agent, it will be still contact agent.

Parameters:

← ***unit_name*** The name of the unit. If the unit does not exist, this function has no effect.

← ***agent_name*** The name of the agent. Agent must be registered in the GAT.

int mgx_agent_new_member (char * *unit_name*, char * *agent_name*, char * *rol_name*)

Function for adding a member to the unit.

The user can add new members to the unit using this function. When an agent is added to a unit it can play some rols. A single agent can play more than one rol in the same unit. Being a member of a unit the agent can see the internal topology of the unit.

Parameters:

← ***unit_ name*** The name of the unit. If the unit does not exist, this function has no effect.

← ***agent_ name*** The name of the agent. It must be running on the platform and so, registered in the GAT. If the agent_name does not exist, this function has no effect.

← ***rol_ name*** The name of the rol associated to the agent. If the pair agent and rol has just been registerd in the unit, this function has no effect.

int mgx_agent_new_unit (char * *unit_ name*, int *routing_ type*)

Function for creating of a new unit. Starting point.

This function allows the user adding a new unit in the platform.

Parameters:

← ***unit_ name*** The name of the new unit. It is associated by the user. The name has to be unique in the platform, i.e., has to be different than other units or agents running on the platform.

← ***routing_ type*** Defines the routing way when the unit receives messages. It is selected by the user and can not be changed. The possibles values aredefined in the previous section*.

→ ***return_ value*** This function returns 1

Ejemplos

En esta sección mostraremos algunas posibilidades que nos ofrece el API de gestión de unidades. Imaginemos que tenemos un sistemas multiagente formado por los agentes de la figura 21

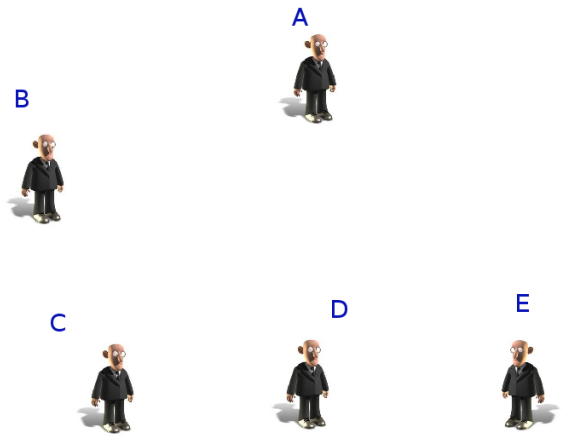

Figura 21: SMA

Entre estos agentes, el agente A ofrece el servicio de validar un usuario en una base de datos. El agente D, a partir de un usuario válido recupera su perfil y sus preferencias. Y el usuario E, a partir de unas preferencias de usuario, selecciona una película.

Nuestro objetivo es integrar toda esta funcionalidad en una unidad organizativa. Para ello, crearemos un agente F que cree la unidad con los agentes que nos interesan y que coordine las conversaciones internas. El objetivo es ofrecer una unidad que recomiendo una película para un usuario. El proceso de creación de la unidad podría ser similar a:

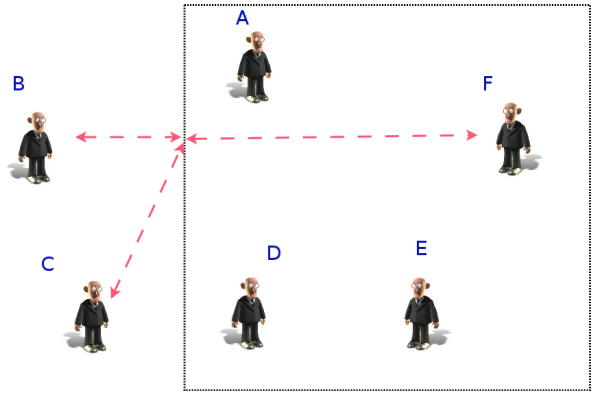

Figura 22: SMA

```
int mgx_main(int argc, char argv[]){

...
*Creamos una nueva unidad*
mgx_agent_new_unit(''recomendacion_pel",OUM_UNIT_ROUTING_TYPE_UNICAST);

*Añadimos los tres agentes con sus roles*
mgx_agent_new_member(''recomendacion_pel",''A",''validador");
mgx_agent_new_member(''recomendacion_pel",''D",''recuperador");
mgx_agent_new_member(‘‘recomendacion_pel",''E",''recomendador");

...
```

Una vez creada la unidad, el agente F recibirá todos los mensajes destinados a la misma (por el hecho de haberle definido un tipo *UNICAST*). Además de aparecer registrada en el *OUM*, podemos registrar un nuevo servicio en el *DF* llamado *recomendacion_peliculas*, de esta manera, el resto de los agentes podrán interactuar con la unidad como si se tratase de un agente individual (figura 22).

Podemos tener un escenario diferente, con los mismos agentes, pero que en este caso queremos que el F no sea el agente que reciba todos los mensajes, sino que interactuen con el agente A. De esta manera, podemos tener un escenario donde el agente F sólo seria el *manager* de la unidad (figura 23, para ello el código necesario seria el siguiente:

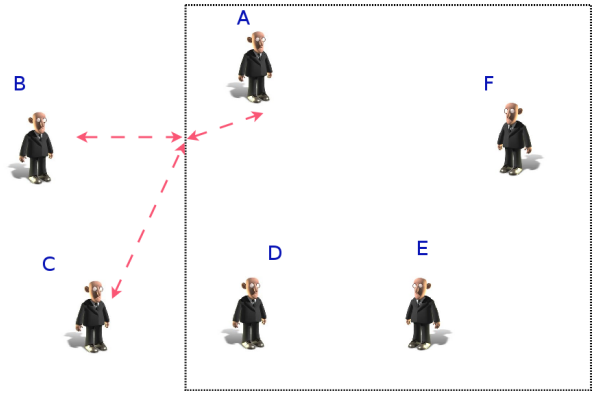

Figura 23: SMA

```
int mgx_main(int argc, char argv[]){

...
*Creamos una nueva unidad*
mgx_agent_new_unit(''recomendacion_pel",OUM_UNIT_ROUTING_TYPE_UNICAST);

*Añadimos los tres agentes con sus roles*
mgx_agent_new_member(''recomendacion_pel",''A",''validador");
mgx_agent_new_member(''recomendacion_pel",''D",''recuperador");
mgx_agent_new_member(``recomendacion_pel",''E",''recomendador");


*Cambiamos el agente de contacto F por A*
mgx_agent_new_contact_agent(''recomendacion_peliculas",''A");
...
```

Observar que en este escenario, puesto que sólo podemos tener un agente de contacto ya que hemos definido un tipo *UNICAST*, no hace falta eliminar el agente *F*, ya que esto se realiza automáticamente cuando añadimos el agente *A*, puesto que no pueden haber dos agentes de contacto.

Otro escenario posible, sería tener una serie de agentes (A, D, E y F), que ofrecen el mismo servicio, por ejemplo, el de consulta de plazas libres de un parking. Estos cuatro agentes pueden estar distribuídos en diversas máquinas y podemos crear una unidad para que este servicio no aparezca replicado en el *DF*. Es decir, uno de los cuatro agentes (por ejemplo el F), creará una unidad y registrará el servicio de *consulta_plazas_libres* en el *DF* asociado a la unidad organizativa. Para que todos los agentes reciban los mensajes

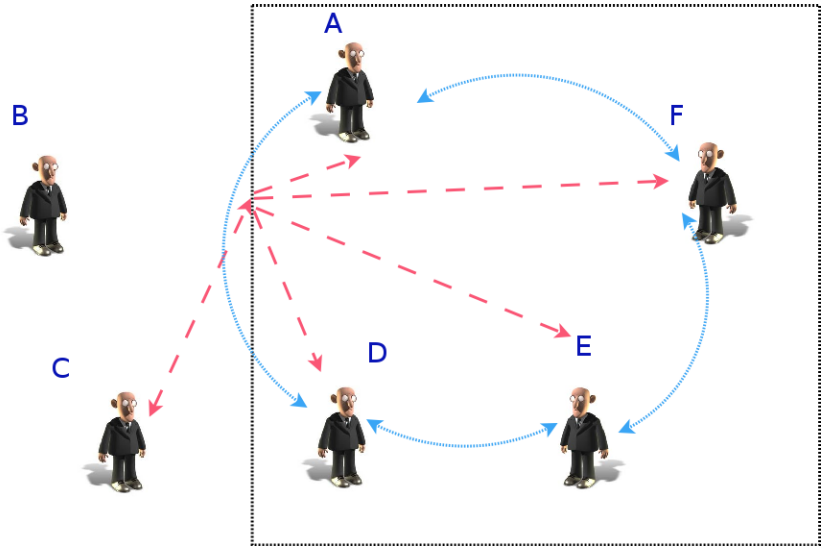

Figura 24: SMA

destinados a la unidad, definiremos un tipo de enrutamiento *MULTICAST* y los cuatro agentes ya se coordinarán para devolver uno de ellos la respuesta (figura 24):

```
int mgx_main(int argc, char argv[]){

...
*Creamos una nueva unidad*
mgx_agent_new_unit(”consulta",OUM_UNIT_ROUTING_TYPE_MULTICAST);

*Añadimos los tres agentes con sus roles*
mgx_agent_new_member(”consulta",”A",”rol_servicio");
mgx_agent_new_member(”consulta",”D",”rol_servicio");
mgx_agent_new_member(”consulta",”E",”rol_servicio");


*Añadimos todos los agentes como contactos*
mgx_agent_new_contact_agent(”consulta",”A");
mgx_agent_new_contact_agent(”consulta",”D");
mgx_agent_new_contact_agent(”consulta",”E");
...
```

Otro ejemplo de uso, podría ser encapsular la funcionalidad de un agente dentro

de una unidad organizativa. Imaginemos que el agente ofrece un servicio de consulta. Podemos registrar una unidad que ofrezca dicho servicio. De esta manera, podemos cambiar el agente que hay dentro de la unidad de una manera transparente al resto de los agentes (figura 25. Bastaría con que el manager de la unidad hiciese algo como:

```
*Añadimos el agente F y elminamos el A*
mgx_agent_new_member(”consulta",”F",”rol_servicio");
mgx_agent_del_member(”consulta",”A",”rol_servicio");
mgx_agent_new_contact_agent(”consulta",”F");
mgx_agent_del_contact_agent(”consulta",”A");
```

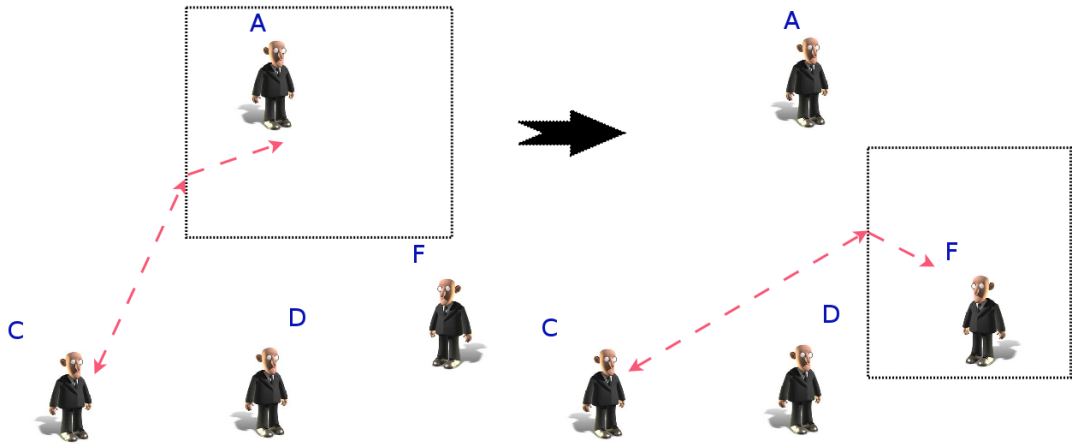

Figura 25: SMA

Conclusiones

La mayoría de las plataformas actuales presentan problemas de baja eficiencia y pobre rendimiento según va aumentando la magnitud del SMA que estamos ejecutando, en el sentido de aumentar el número de hosts, agentes y el tráfico entre los mismos. Esta pobre escalabilidad hace que sea un impedimento para construir sistemas complejos compuestos por miles de agentes. Desde el punto de vista de la tecnología de SMA, el concepto de que estos sistemas estén implementados en entornos distribuídos, abiertos y con un gran volumen de carga, no se plantea como una posibilidad, más bien como un requerimiento.

Actualmente, existe una gran cantidad de entornos de ejecución para SMA. Diversos grupos de investigación del área, centran sus estudios en diseñar nuevas plataformas que aporten, posiblemente alguna característica diferente, pero basándose en la arquitectura interna que está presente en las plataformas ya existentes. Este hecho hace que prácticamente todas ellas ofrezcan unas tasas de pobre rendimiento y baja escalabilidad. Nuestra experiencia analizando plataformas nos ha permitido confirmar que la gran mayoría de diseños actuales presentan una gran degradación en la eficiencia según se va aumentando la magnitud del SMA que estamos ejecutando.

Teniendo en cuenta estos estudios, hemos planteado el diseño inicial de la plataforma Magentix, la cual tiene una arquitectura interna innovadora ya que está diseñada en C sobre el SO. El simple hecho de acercar este diseño hace que la plataforma sea mucho más escalable, presentando una menor degradación en la eficiencia que el resto de plataformas actuales.

La plataforma se encuentra en un estado de implementación donde se ofrecen las funcionalidades más básicas que podamos requerir para desarrollar SMA así como otras funcionalidades que no están presentes en la mayoría de las plataformas. El hecho de poder ejecutar SMA de gran tamaño en Magentix, nos introduce la posibilidad de desarrollar dichos SMA mediante el concepto de organizaciones, el cual nos ofrece un entorno que

nos ayuda a simplificar, estructurar y desarrollar de una manera más flexible a la vez que potente, los SMA. Las organizaciones son especialmente interesantes cuando hablamos de sistemas complejos y abiertos.

En este trabajo se ha planteado un diseño para incorporar el concepto de organizaciones de agentes en la plataforma Magentix, basándonos en la arquitectura de la misma. Primeramente se ha realizado un estudio previo de los requisitos de las organizaciones de agentes, así como de algunas plataformas que soportan de alguna u otra manera este concepto. Hemos presentado una forma sencilla para incluir algunos conceptos que deben estar presentes en las organizaciones y se ha dejado una línea abierta para continuar ampliando la funcionalidad. El diseño presentado ha sido centrándonos en las comunicaciones e interacciones entre agentes y grupos de agentes, a los que hemos llamado unidades organizativas. Estas unidades presentan algunas de las características presentes en las organizaciones. El concepto de unidad organizativa no sólo nos restringe a diseñar organizaciones de agentes, sinó que abre un nuevo abanico de posibilidades en las interacciones agente-grupo de agentes, como hemos podido ver en las secciones anteriores.

Hemos planteado el concepto de unidad organizativa como una estructura que nos permitirá crear unidades más complejas. Estas unidades son agrupaciones de agentes que actuan como una única entidad de cara a los agentes externos. La interacción de agentes individuales con estas unidades es totalmente transparente ya que se ha modificado el mecanismo de entrega de mensajes para dar cabida a una diversidad de alternativas que nos plantea el concepto de unidad o agrupación de agentes. Estas unidades requieren de un nuevo servicio que hemos llamado *OUM* y que gestionará el funcionamiento de las unidades de la plataforma de una manera distribuída. Para ofrecer una programación más fácil de cara al usuario del sistema, se ha presentado una API básica para el manejo de unidades. Esta propuesta inicial nos permite incorporar una nueva alternativa para desarrollar SMA en la plataforma Magentix, dejando abierta una línea de desarrollo para ampliar esta funcionalidad.

El concepto de organización de agentes requiere de muchas características, pero a partir del diseño inicial se podrán incorporar en un trabajo el resto de ellas como son la definición de reglas o normas, de manera que se cumplan las pautas que definamos para las organizaciones, los objetivos globales que definamos para las organizaciones, los planes o estrategias asociados a los roles, los protocolos de interacción definidos entre los distintos roles así como lograr una mayor interacción de los agentes con las organizaciones, ampliando las posibilidades de publicitación e interacción de servicios. También necesitamos desarrollar ejemplos de aplicaciones reales dónde realmente sea interesante el uso de aplicaciones, así como realizar pruebas de rendimiento para medir diversos parámetros.

Bibliografía

[1] Fipa rdf. `http://www.fipa.org/specs/fipa00011/XC00011B.html`.

[2] Magentix, multiagent platform integrated in linux (magentix). http://www.dsic.upv.es/users/ia/sma/tools/Magentix/index.html.

[3] Web de rdf. `http://www.w3.org/RDF/`.

[4] Juan M. Alberola. Analisis completo de la plataforma de agentes agentscape. Internal report, GTI Magentix Internal Report, 2006.

[5] Juan M. Alberola, Luis Mulet, Jose M. Such, Ana Garcia-Fornes, Agustin Espinosa, and Vicent Botti. Operating system aware multiagent platform design. In *Proceedings of the Fifth International Workshop on Multi-Agent Systems (EUMAS-2007)*, pages 658–667, 2007.

[6] Carter J. Ghorbani A. Bitting, E. Multiagent system development kit: An evaluation. In *Proceedings of Communication Networks and Services Research Conference, May 15-16, pp. 80-92, Moncton, New Brunswick, Canada, 2003.*

[7] K. Burbeck, D. Garpe, and S.Ñadjm-Tehrani. Scale-up and performance studies of three agent platforms. In *IPCCC 2004.*

[8] D. Camacho, R. Aler, C. Castro, and J. M. Molina. Performance evaluation of zeus, jade, and skeletonagent frameworks. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on.*

[9] Tomiak D. Gawinecki M. Karczmarek P. Chmiel, K. Testing the efficency of jade agent platform. In *Proceedings of the ISPDC/HeteroPar'04, 49-56.*

[10] E. Cortese, F.Quarta, and G. Vitaglione. Scalability and performance of jade message transport system. *EXP*, 3:52–65, 2003.

[11] V. Julian E. Argente and V. Botti. From human to agent organizations. In *CoOrg-05: Proceedings of the First International Workshop on Coordination and Organisation*, 2005.

[12] FIPA. *FIPA ACL Message Structure Specification*. FIPA, 2001.

[13] Nguyen T. Giang and Dang T. Tung. Agent platform evaluation and comparison. 2002.

[14] L. C. Lee, D. T. Ndumu, and P. De Wilde. The stability, scalability and performance of multi-agent systems. *BT Technology Journal*, 16:94–103, 1998.

[15] L.C. Lee, Nwana, H.S., Ndumu, D.T., and P. De Wilde. The stability, scalability and performance of multi-agent systems. In *BT Technology J.*, volume 16, 1998.

[16] Luis Mulet. Estudio interno de la plataforma multiagente madkit. Internal report, GTI Magentix Internal Report, 2006.

[17] Luis Mulet, Jose M. Such, Juan M. Alberola, Vicent Botti, Agustin Espinosa, Ana Garcia-Fornes, and Andres Terrasa. Performance evaluation of open-source multi-agent platforms. In *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06)*, pages 1107–1109. Association for Computing Machinery, Inc. (ACM Press), 2006.

[18] H.Ñwana. Negotiation strategies: An overview. Internal Report 14, BT Laboratories, 1994.

[19] Pierre-Michel Ricordel and Yves Demazeau. From analysis to deployment: a multi-agent platform survey. In *ESAW'00, Engineering Societies in the Agents' World.*

[20] Elhadi Shakshuki. A methodology for evaluating agent toolkits. In *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume I*, pages 391–396, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Elhadi Shakshuki and Yang Jun. Multi-agent development toolkits: An evaluation. *Lecture Notes in Computer Science*, 3029:209–218, 2004.

[22] D. Sislak, M. Rehak, M. Pechouce, M. Rollo, and D. Pavlicek. A-globe: Agent development platform with inaccessibility and mobility support, 2005.

[23] Jose M. Such. Estudio de la plataforma de agentes jade. Internal report, GTI Magentix Internal Report, 2006.

[24] Jose M. Such, Juan M. Alberola, Luis Mulet, Agustin Espinosa, Ana Garcia-Fornes, and Vicent Botti. Large-scale multiagent platform benchmarks. In *Federated Workshops, Multi-Agent Logics, Languages, and Organisations (MALLOW-07)*, 2007.

[25] P. Vrba. Java-based agent platform evaluation. In *Proceedings of the HoloMAS 2003*, pages 47–58, 2003.