Tesina de Máster

Seguridad en Sistemas Multiagente

Jose Miguel Such Aparicio
jsuch@dsic.upv.es

Dirigido por:
Agustín Espinosa Minguet
Ana García Fornés

Septiembre de 2008

Índice

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

Los Sistemas Multiagente (SMA) están formados por entidades software (agentes), con un cierto grado de inteligencia, que interactúan entre sí para conseguir sus objetivos. La interacción se efectúa por medio de mecanismos de comunicación, que permiten a los agentes realizar peticiones a otros agentes.

Diversos grupos de investigación han invertido grandes esfuerzos en desarrollar plataformas que den soporte a los SMA, cada una de ellas con sus propias características de ejecución e interconexión de los agentes. En haras de estandarizar sobretodo las comunicaciones para garantizar la interoperabilidad entre agentes de plataformas diferentes, existe la Foundation for Intelligent Physical Agents (FIPA) [1].

Los estudios realizados en [2], [3], [4], [5], [6], [7] [8], [9], [10] 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 ofrecen, en mayor o menor medida, excesiva degradación en el tiempo de respuesta cuando la carga de la misma es alta.

Teniendo en cuenta todos estos estudios se ha desarrollado la plataforma de agentes MAGENTIX. Esta plataforma ha sido desarrollada por el Grupo de Tecnología Informática - Inteligencia Artificial del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia, y está implementada en el lenguaje de programación C. Se trata de una plataforma para dar soporte a SMA de manera distribuida, por tanto,

se puede ejecutar sobre diversos ordenadores, manteniendo la información necesaria de forma replicada. Su diseño está pensado en haras de ofrecer un alto rendimiento y un alto grado de escalabilidad.

Un aspecto clave en SMA resulta ser la seguridad en las interacciones, el poder garantizar una comunicación segura y compatible entre los agentes participantes. Es decir, que los mensajes que se intercambian en cada interacción estén provistos de integridad, confidencialidad e irrefutabilidad. Así pues, se hablará de cómo se ha resuelto el problema de las comunicaciones seguras en Magentix.

Los trabajos referentes a seguridad dentro del campo de los SMA son todavía bastante escasos. La seguridad se está convirtiendo en un aspecto importantísimo dentro del campo de la interoperabilidad entre agentes y los SMA en general, ya que de forma que las aplicaciones basadas en agentes autónomos inteligentes y SMA van creciendo, va apareciendo la necesidad de entender realmente los riesgos asociados a su uso. Un comportamiento incorrecto o inapropiado de un agente puede tener efectos no deseados, incluyendo: pérdida de dinero, pérdida de datos, y incluso en ciertas aplicaciones puede llegar a peligrar la integridad de un humano o del sistema en cuestión. Por tanto, la seguridad se vuelve un factor importantísimo a tener en cuenta cuando se desarrollan estos tipos de sistemas, y en parte, la falta de seguridad en estas aplicaciones es una de las causas de porqué la tecnología de SMA se está implantando de forma lenta en la industria.

Algunas plataformas de agentes empiezan a tener muy en cuenta aspectos de seguridad. Jade [11], SECMAP [12], Tryllian ADK [13], CAPA [14], Cougaar [15], SeMoA [16] and Voyager [17] son algunas de las plataformas de agentes que tienen en cuenta aspectos de seguridad. Todas estas plataformas ofrecen, autenticación, integridad y confidencialidad. Además algunas de ellas ofrecen mecanismos de control de acceso a recursos.

Por ejemplo, la plataforma Jade, que actualmente es la más utilizada a la hora de desarrollar SMA, incluye un add-on de seguridad que proporciona autenticación, control de acceso mediante Java/JAAS, y también proporciona integridad y confidencialidad utilizando criptografía. Como cabe esperar al introducir controles de seguridad va a haber una sobrecarga en el tiempo de respuesta de la plataforma como queda claro en [18], trabajo en el que se presentan benchmarks para plataformas que den soporte a SMA a gran escala y justamente el ejemplo que hay es una comparación de Jade normal

junto con Jade con el add-on de seguridad.

El presente trabajo trata de añadir a Magentix aspectos de seguridad para lograr asegurar las interacciones entre agentes pertenecientes a la plataforma. De lo dicho anteriormente sobre Jade y su add-on de seguridad se puede inferir que si se quieren asegurar las interacciones en Magentix se va a introducir una sobrecarga inevitable. De todas formas, en este trabajo se van a presentar las alternativas barajadas para asegurar Magentix y la elección final, siempre teniendo en cuenta los aspectos sobre los que se basa Magentix, que como se ha comentado, son el rendimento y el alto grado de escalabilidad pretendidos para poder dar soporte a SMA reales, y que por tanto serán complejos con un alto número de agentes e interacciones entre ellos.

El documento continúa con una explicación de la arquitectura interna de la plataforma Magentix en la sección 2. En la sección 3 se presentan las características de seguridad que se quieren conseguir en Magentix, las alternativas que se plantean y la elección realizada justificándola en base al propio diseño interno de Magentix. La sección 4 muestra el control de acceso necesario para cumplir las características de seguridad deseadas. La sección 5 presenta la integración que se ha hecho del protocolo Kerberos para así crear lo que he llamado Magentix-S (versión segura de Magentix). La sección 6 ilustra el estudio de la pérdida de eficiencia que se obtiene en Magentix-S respecto con Magentix. La sección 7 presenta una guía de instalación y configuración de Magentix-S. Finalmente, en la sección 8 se presentan las conclusiones y las posibles ramas de investigación futuras en este campo.

2. La Plataforma de Agentes Magentix

2.1. Introducción

La plataforma MAGENTIX (una plataforma Multi-AGENTe integrada en LINUX) está desarrollada en C sobre el Sistema Operativo (SO) Linux ([19], [20], [21]). 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.

2.2. 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 ha sido implementada en el

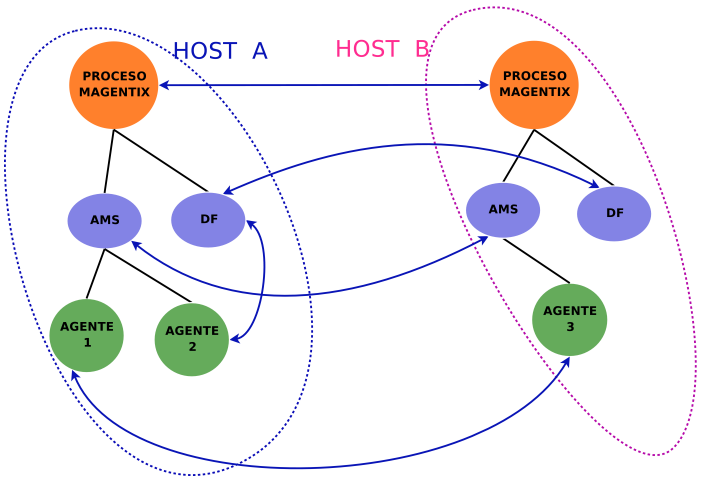

Figura 1: Estructura de la plataforma

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:

```
./magentix
```

ejecutada en el directorio donde tenemos los fuentes de la plataforma, mediante un intérprete de órdenes como puede ser el *Shell* de Linux. Este proceso tendrá como UID y GID efectivos los del usuario que ha ejecutado la órden anterior para lanzar la plataforma.

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.

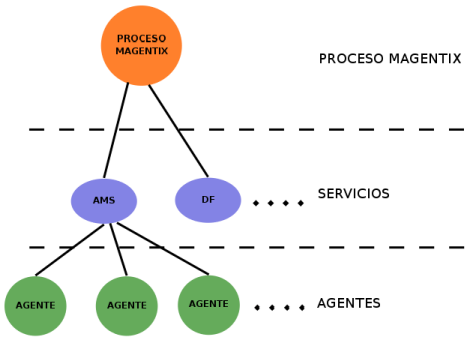

Figura 2: Árbol de procesos

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.

2.3. 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:

```
./magentix
```

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:

```
./magentix -d NOMBRE_HOST_PRINCIPAL
./magentix -a DIRECCIÓN_IP_HOST_PRINCIAPL
```

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. Evidentemente, esta IP o nombre de host debe corresponder a un host que ya esté ejecutando Magentix y que además, haya sido el primer host de la plataforma. Intentar unirse a una plataforma Magentix especificando una IP o un nombre de host que no corresponde al host principal de esa plataforma, no tendrá ningún efecto.

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.

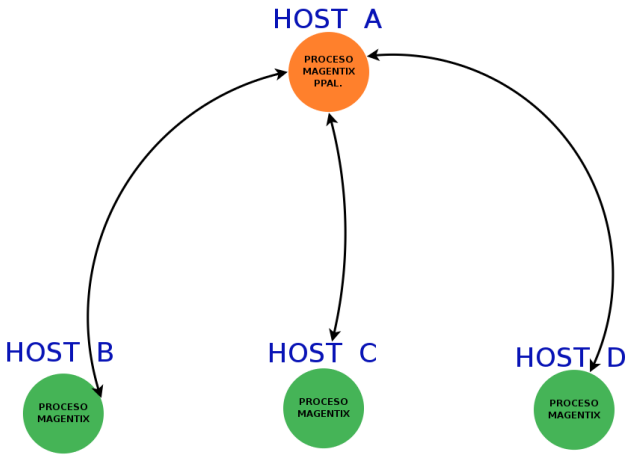

Figura 3: Distribución de los procesos *magentix*

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 Magentix, 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.

2.4. Servicios de la plataforma

2.4.1. 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 2.3. 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 2.2, 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 correspondiente, 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 2.1, cada agente dispondrá de un puerto TCP para comunicarse con otros agentes. Por tanto, la información de contacto almacenada en esta tabla es la direc-

ción 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). Así, si se requiere información de este tipo sobre algún agente que se esté ejecutando en otro host, se ha de solicitar dicha información al proceso *ams* pertinente. 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 2.3). 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 2.3.

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, ejectuaremos el proceso *new_agent*, proporcionando como argumentos obligatorios el nombre del agente y la ruta del binario:

```
./new_agent nombre ruta
```

Este proceso, 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 secundaria que puede ser requerida en algún momento por algún proceso *ams* pero nunca por agentes, y además, la frecuencia con la que se requiere esta información es mucho menor que la que se encuentra en la *GAT*.

2.4.2. El Servicio DF (Directory Facilitator)

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, 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 2.4.1. 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. Tal y como ocurre con la tabla GAT 2.4.1, esta tabla se encuentra mapeada en memoria compartida, de manera que cuando un agente requiera buscar un servicio, podrá consultarlo en esta tabla. El proceso *df* tendrá acceso de lectura y escritura a esta tabla mientras que los agentes sólo podrán leer la información.

A diferencia del servicio AMS donde las inserciones y borrados de la tabla GAT eran hechas automáticamente por los procesos *ams*, las inserciones y borrados de la tabla GST implican que sean los agentes los que los soliciten. Cuando un agente quiera dar de alta o desregistrar algún servicio suyo, se comunicará con su *df* local para transmitirle esa petición. El *df* local ya se encargará de realizar las gestiones indicadas en el párrafo anterior. En cambio, si un agente requiere de los servicios de otro agente, puede realizar una búsqueda en la GST sin necesidad de recurrir al *df* ya que se trata de una operación exclusivamente de lectura.

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

2.4.3. 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 comunica-

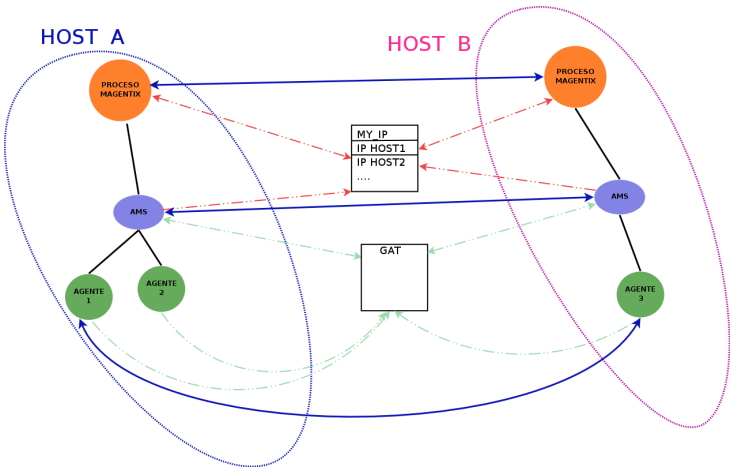

Figura 4: Arquitectura distribuída de la plataforma

ciones 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. 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).

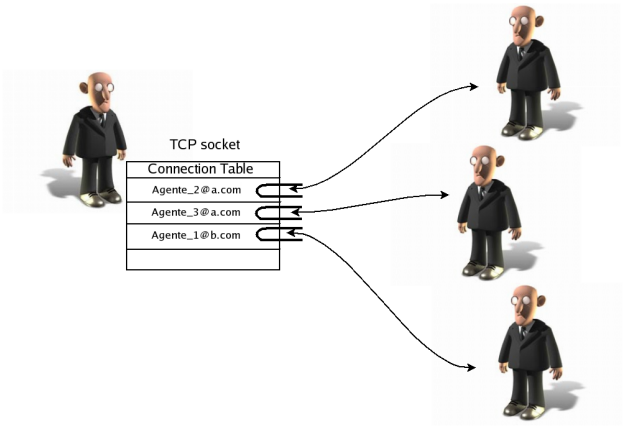

Figura 5: Modelo de comunicación

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.

2.5. 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. Cada agente también tiene acceso de lectura a la tabla GST, con lo que la búsqueda de servicios también es rápida sin que aparezca ningún cuello de botella por ningún elemento que centralice estas gestiones. El hecho de que estas tablas se encuentren replicadas en cada host por los servicios distribuídos AMS y 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.

2.5.1. 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).

2.5.2. 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

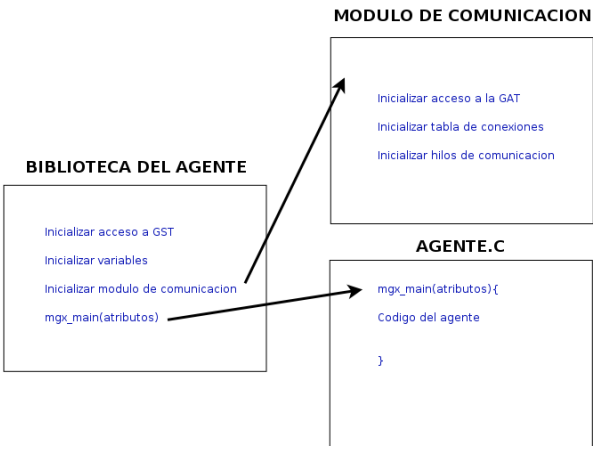

Figura 6: Inicialización de los módulos del agente

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

2.5.3. Estructura de los mensajes

Los mensajes que se intercambian los agentes Magentix puede ser cualquier cadena de bytes, pero además, existe una API de gestión de mensajes la cuál ha sido implementada siguiendo el estándard FIPA.

En la biblioteca *mgx_message* se detalla la estructura de un mensaje Magentix el cuál está compuesto por los campos definidos en FIPA. Además se proporcionan una serie de funciones que componen el API para la gestión de dichos campos.

Se define la estructura de un mensaje Magentix como del tipo *mgx_message_t*. Esta estructura representa por defecto, una zona contígua de memoria de 4 Kb. En los primeros bytes de estra estructura se define la cabecera del mensaje como una serie de atributos enteros del mensaje (bytes totales del mensaje, último byte en términos de posiciones relativas, etc.), así como una serie de punteros a las zonas de memoria donde se encuentran los campos definidos por FIPA y que corresponden a ese mensaje concreto.

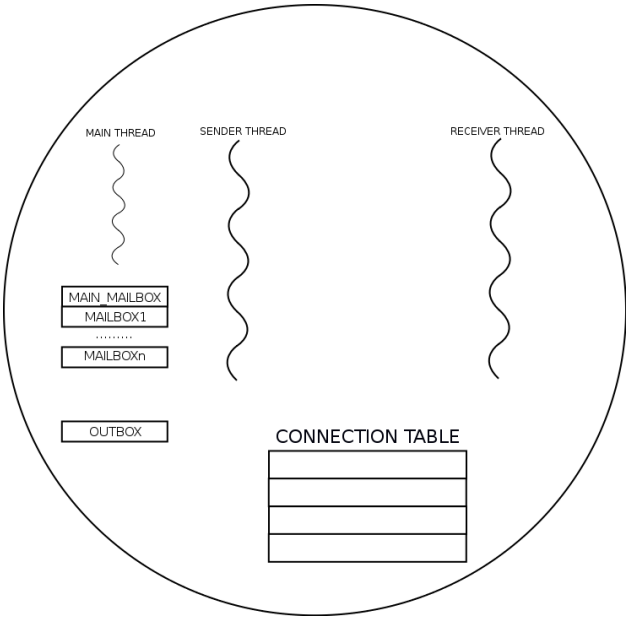

Figura 7: Estructura interna de un agente Magentix

Como podemos ver en la Figura 8, en la cabecera existen las posiciones de memoria de los campos del mensaje dentro de ese espacio contiguo de memoria de 4 Kb. Así podemos ver como todos los campos definidos por FIPA, han de quedar en ese espacio de memoria, contemplando incluso, campos múltiples como el de *receptor*. Esta estructura de 4 Kb correspondería a lo que se define en FIPA como la cabecera del mensaje. Si se requiriese de una longitud mayor para ubicar todos los campos del mensaje, se debería modificar la biblioteca para definir un trozo de memoria contígua mayor.

El hecho de tener un espacio de memoria contígua donde ubicar todos los campos de la cabecera del mensaje, nos permitirá enviar toda esa información a otro agente, sin necesidad de reubicar el mensaje antes de enviarlo. Observamos también que el contenido del mensaje va en una zona de memoria aparte, ya que es un atributo del mensaje cuya longitud es más impredecible y puede ocupar más de 4 Kb.

Como hemos explicado anteriormente, el *sender_thread* de cada agente, será el encargado de enviar los mensajes del mismo. Podemos ver que el mensaje en un principio estará fragmentado en dos partes: la parte de la

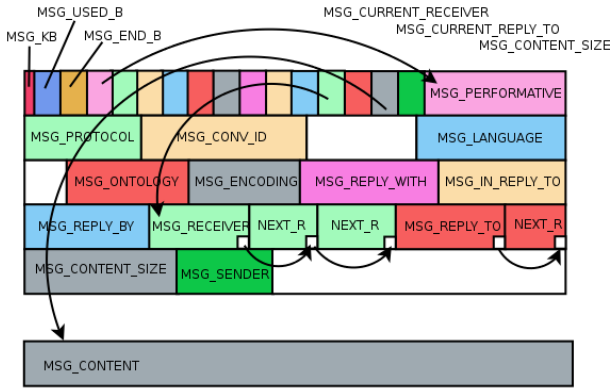

Figura 8: Estructura de un mensaje Magentix

cabecera, y la parte del contenido. El *sender_thread*, una vez ya sabe el socket por el que enviará el mensaje, reubica la cabecera y el contenido del mensaje en una zona contígua de memoria para realizar una única escritura en el socket. Esta decisión ha sido tomada porque consume un tiempo de ejecución menor que enviando los dos fragmentos del mensaje por separado mediante dos escrituras en el socket.

Evidentemente, si la cabecera no ocupa los 4 Kb máximos definidos por defecto, se enviará una cantidad de memoria menor. Pero los posibles vacíos de memoria que pueden crear entre los campos del mensaje (como podemos ver en la Figura 8) no se reubicarán antes de enviar el mensaje. Estos vacíos pueden quedar simplemente por la gestión que realizan de la zona contígua de memoria las funciones del API, según su implementación.

A la hora de la recepción de cualquier mensaje, el *receiver_thread* reservará la memoria correspondiente a la longitud del mensaje que se va a recibir, y reubicará correctamente el mensaje recibido en una estructura del tipo *mgx_message_t*, para que la gestión del mismo mediante las funciones del API sea más práctica.

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

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

Las dos funciones envían el mensaje que se le pasa por parámetro. 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_message_t *message);
mgx_comm_try_receive(mgx_mailbox_id_t mailbox, mgx_message_t *message);
mgx_comm_timed_receive(mgx_mailbox_id_t mailbox, mgx_message_t *message, struct
                  timespec timeout);
```

Las tres funciones recibirán un mensaje del Mailbox indicado como primer parámetro. Internamente el hilo *receiver_thread* se encargará de reservar la memoria necesaria para el mensaje recibido. 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*.

Además, se dispone de toda la funcionalidad para manejar los campos de los mensajes con funciones de inserción de campos, borrados, consulta y consulta de tamaños. Un ejemplo podría ser:

```
int mgx_message_set_protocol (mgx_message_t *message, char *protocol);
int mgx_message_delete_protocol (mgx_message_t *message);
int mgx_message_get_protocol (mgx_message_t *message, char *protocol);
int mgx_message_get_protocol_size (mgx_message_t *message, int *size);
```

3. Asegurando las Interacciones

Antes de pasar directamente a hablar de seguridad en las interacciones entre agentes, y concretamente el soporte de seguridad elegido para las interacciones de los agentes Magentix, se va a hacer una introducción general de lo que se considera en informática un sistema seguro.

Aunque no existe un sistema informático 100 % seguro, las características que debe ofrecer un sistema informático seguro son:

- **Fiabilidad**. El sistema se comporta como debe.
- **Integridad**. La información no puede ser modificada por quien no está autorizado.
- **Confidencialidad**. La información sólo debe ser legible para los autorizados.
- **Irrefutabilidad**. El sistema impide poder negar la autoría.
- **Disponibilidad**. El sistema es accesible cuando se necesita.
- **Consistencia**. El sistema se comporta como debe a lo largo del tiempo.

En la escasa y joven literatura referente a seguridad en Sistemas Multiagente, el concepto de seguridad se divide en dos términos, que en inglés se corresponden con *safety* y *security*. La traducción para los dos términos es la misma en español, pero realmente, tanto en congresos como en las publicaciones tienen matices diferentes:

- **Safety**: los agentes tienen que ser suficientemente robustos para seguir mostrando el mismo comportamiento predefinido ante eventos inesperados o alteraciones técnicas. Por tanto, engloba las características de fiabilidad, consistencia y disponibilidad.
- **Security**: los agentes tienen que resistir efectos potencialmente dañinos debidos a comportamientos maliciosos por parte de otros agentes. Englobando por tanto: integridad, confidencialidad e irrefutabilidad.

Actualmente para asegurar las comunicaciones en la red se trata de garantizar la integridad, confidencialidad y la irrefutabilidad. Por el diseño de Magentix, todas las interacciones entre agentes se van a realizar mediante una interfaz de red (local o remota). Por tanto, para asegurar las interacciones en Magentix se han de garantizar estas tres características. Para ello se han estudiado distintas alternativas que se discuten en los apartados siguientes llegando a una conclusión sobre la más adecuada para Magentix.

3.1. Alternativas

3.1.1. IPSEC

IPsec (la abreviatura de Internet Protocol Security) [22] es una extensión al protocolo IP que añade cifrado fuerte para permitir servicios de autenticación y, de esta manera, asegurar las comunicaciones a través de dicho protocolo. Inicialmente fue desarrollado para usarse con el nuevo estándar IPv6, aunque posteriormente se adaptó a IPv4.

IPsec actúa a nivel de capa de red, protegiendo y autenticando los paquetes IP entre los equipos participantes en la comunidad IPsec. No está ligado a ningún algoritmo de cifrado o autenticación, tecnología de claves o algoritmos de seguridad específico. Es más, IPsec es un marco de estándares que permite que cualquier nuevo algoritmo sea introducido sin necesitar de cambiar los estándares.

3.1.2. SSL/TLS

Secure Sockets Layer (SSL) [23] y Transport Layer Security (TLS) [24], son protocolos criptográficos que proporcionan autenticación y privacidad de la información entre extremos sobre Internet. TLS es la estandarización hecha por IETF de SSL v3.1. Habitualmente, sólo el servidor es autenticado (es decir, se garantiza su identidad) mientras que el cliente se mantiene sin autenticar; la autenticación mutua requiere un despliegue de infraestructura de claves públicas (o PKI) para los clientes.

SSL implica una serie de fases básicas:

1. Negociar entre las partes el algoritmo que se usará en la comunicación
2. Intercambio de claves públicas y autenticación basada en certificados digitales
3. Cifrado del tráfico basado en cifrado simétrico

Como TLS es la última versión de estos protocolos será el que comparemos con las otras alternativas.

3.1.3. Kerberos

Kerberos [25] es un protocolo de autenticación de red. Está diseñado para proporcionar autenticación fuerte para aplicaciones basadas en el paradigma cliente/servidor utilizando criptografía de clave secreta. Desde el MIT (Massachusets Institute of Technology) se puede obtener una implementación libre de este protocolo bajo una licencia similar a la utilizada en los sistemas operativos BSD y el sistema de ventanas X. Además, también está disponible en muchos productos comerciales.

El protocolo Kerberos utiliza criptografía fuerte de forma que un cliente puede probar su identidad a un servidor (y viceversa) a través de una conexión de red insegura. Una vez que el cliente y el servidor han probado su identidad, además pueden encriptar todas sus comunicaciones para asegurar la privacidad y la integridad de los datos que intercambian.

3.2. Discusión y Elección

Las tres alternativas presentadas ofrecen las características deseadas, aunque de distinto modo. IPSec por ejemplo, al actuar a nivel de capa de red no es consciente de los protocolos de mayor nivel que viajan en los datagramas IP, con lo que no sería capaz de distinguir entre agentes Magentix, puesto que cada uno de ellos escucha en un puerto TCP diferente, con lo que la autenticación se realizaría a nivel de host perteneciente a la plataforma y no a nivel de agente. Por tanto, de las tres alternativas es la única que no cubre los requerimientos de seguridad para Magentix.

Tanto TLS como Kerberos, al trabajar a nivel de transporte y a nivel de aplicación respectivamente, si que permiten autenticación de agentes. En principio, al actuar TLS en la capa de transporte y Kerberos en la capa de aplicación, debería ser mucho más fácil integrar en Magentix TLS que Kerberos. Kerberos no se encarga del envío de datos, eso lo deja en manos del usuario, mientras que en TLS se reemplazan las funciones de sockets convencionales por sus equivalentes seguras. Además, usando TLS, la encriptación es transparente al programador, es decir, no hay una función para explícitamente cifrar lo que se va a enviar. En cambio, en Kerberos, es el programador el que tiene que llamar a funciones del API para cifrar y descifrar. Por tanto, integrar Kerberos en Magentix va a requerir un mayor esfuerzo de programación.

En la sección 2.4.3, se explica como está implementado el servicio de mensajería entre agentes en Magentix. Como allí se detalla, existe una cache de conexiones abiertas con otros agentes con el objetivo de mantener abiertas las conexiones con los agentes que hace menos tiempo que se ha hablado con ellos, y que por tanto, probablemente, se volverá a interaccionar con ellos en breve. Esta implementación se debe a el número máximo de sockets que puede tener abiertos una aplicación están limitados por el Sistema Operativo, y como los SMA son sistemas con un alto grado de interacciones, puede que un mismo agente esté conversando con muchos más agentes que conexiones permita el Sistema Operativo.

Llegado a este punto, por la propia implementación de Magentix, Kerberos ofrece una ventaja fundamental: al actuar a nivel de aplicación, el mismo contexto de seguridad puede ser usado en distintas conexiones. Por tanto, aunque una conexión se cierre por haber muchas conversaciones, el contexto de seguridad asociado se puede guardar para que cuando el agente en cuestión vuelva a conversar con el agente con el que ha cerrado la conexión no haga falta renegociar el contexto de seguridad. En TLS, esto no es posible, ya que se tiene que realizar la negociación del contexto de seguridad en cada conexión, y justamente, este es el proceso que más tiempo consume en TLS. Además, aunque en TLS se puede renegociar a partir de un contexto anterior, se pierde un tiempo similar al de creación de uno nuevo. Esto a priori podría no tener mucha importancia en otras soluciones software, pero los SMA reales son sistemas complejos con un gran número de agentes y un alto grado de interacciones entre ellos, con lo que teóricamente se podría obtener una gran mejora en eficiencia evitando al máximo posible neogiaciones de

contextos de seguridad.

Otra diferencia significativa entre las dos alternativas resulta ser el tipo de criptografia utilizada. Mientras TLS utiliza tanto criptografía asimétrica como simétrica, Kerberos sólo utiliza criptografía simétrica. En criptografía simétrica se usa una misma clave para cifrar y para descifrar mensajes, mientras que en criptografía asimétrica se usa una clave para cifrar y otra clave para descifrar. En criptografía simétrica los algoritmos son menos complejos y las claves requieren menos bits, con lo que resulta más eficiente que la criptografía asimétrica. En cambio, la criptografía simétrica presenta el problema de distribución de claves inexistente en criptografía asimétrica, puesto que no se necesita un secreto compartido entre el que cifra y el que descifra.

Por tanto, y ya que la plataforma Magentix tiene muy en cuenta aspectos de eficiencia y escalabilidad, parece que la mejor opción debería ser Kerberos ya que solamente utiliza criptografía simétrica que resulta ser más eficiente que la asimétrica. Los problemas de eficiencia de TLS se ven claramente expuestos en el trabajo de Coarfa y otros [26], en donde se muestra que al utilizar TLS para asegurar un servidor web se incrementa la sobrecarga en un factor del 3.4 al 9. Como dato relevante, cabe destacar que el coste de las operaciones de criptografía asimétrica representa del 20 % al 58 % del total. De esta forma, otros trabajos tratan de modificar TLS para añadir claves compartidas con anterioridad para eliminar estas operaciones de criptografía asimétrica. En esta línea el trabajo de Kuo y otros [27] presenta una comparación entre TLS convencional contra TLS modificado para utilizar claves pre-compartidas (es decir, compartidas con anterioridad). Los resultados muestran que se obtiene mejor rendimiento con claves pre-compartidas, pero les surge el problema de distribución de claves. Kerberos tiene el problema de distribución de claves solucionado, y además solamente utiliza criptografía simétrica, con lo que parece clara la elección para una plataforma de alto rendimiento como pretende ser Magentix.

Por todo el razonamiento anterior, entendemos que la mejor opción para la plataforma de agentes Magentix, debido principalmente a las características de diseño de la misma, es Kerberos.

3.3. El protocolo kerberos

Básicamente el funcionamiento del protocolo de autenticación de red Kerberos es el siguiente: el cliente se autentica a sí mismo contra el AS (Authentication Server) así demuestra al TGS (Ticket Granting Server) que está autorizado para recibir un ticket de servicio. Así, recibe un ticket con el que demostrar al SS (Service Server) que ha sido autorizado para hacer uso del servicio kerberizado.

A continuación se describen los pasos que se siguen en el protocolo. También se pueden ver los pasos gráficamente en la figura 9:

1. Un usuario ingresa su nombre de usuario y password en el cliente.
2. El cliente genera una clave hash a partir del password y la usará como la clave secreta del cliente.
3. El cliente envía un mensaje en texto plano al AS solicitando servicio en nombre del usuario.
4. El AS comprueba si el cliente está en su base de datos. Si es así, el AS envía dos mensajes al cliente:
   a) Mensaje A: Client/TGS session key cifrada usando la clave secreta del usuario.
   b) Mensaje B: Ticket-Granting Ticket (que incluye el ID de cliente, la dirección de red del cliente, el período de validez el Client/TGS session key) cifrado usando la clave secreta del TGS.
5. Una vez que el cliente ha recibido los mensajes, descifra el mensaje A para obtener el client/TGS session key. Esta clave de sesión se usa para comunicarse con el TGS. (El cliente no puede descifrar el mensaje B pues para cifrar éste se ha usado la clave del TGS). En este momento el cliente ya se puede autenticar contra el TGS.
6. Cuando solicita un servicio el cliente envía los siguientes mensajes al TGS:
   a) Mensaje C: Compuesto del Ticket-Granting Ticket del mensaje B y el ID del servicio solicitado.

   b) Mensaje D: Autenticador (compuesto por el ID de cliente y una marca de tiempo), cifrado usando el client/TGS session key.

7. Cuando recibe los mensajes anteriores, el TGS descifra el mensaje D (autenticador) usando el client/TGS session key y envía los siguientes mensajes al cliente:

   a) Mensaje E: Client-to-server ticket (que incluye el ID de cliente, la dirección de red del cliente, el período de validez y una Client/Server session key) cifrado usando la clave secreta del servicio.

   b) Mensaje F: Client/server session key cifrada usando el client/TGS session key.

8. Cuando el cliente recibe los mensajes E y F, ya tiene suficiente información para autenticarse contra el SS. El cliente se conecta al SS y envía los siguientes mensajes:

   a) Mensaje E del paso anterior.

   b) Mensaje G: un nuevo Autenticador que incluye el ID de cliente, una marca de tiempo y que está cifrado usando el client/server session key.

9. El SS descifra el ticket usando su propia clave secreta y envía el siguiente mensaje al cliente para confirmar su identidad:

   a) Mensaje H: la marca de tiempo encontrada en el último Autenticador recibido del cliente más uno, cifrado el client/server session key.

10. El cliente descifra la confirmación usando el client/server session key y chequea si la marca de tiempo está correctamente actualizada. Si esto es así, el cliente confiará en el servidor y podrá comenzar a usar el servicio que este ofrece.

11. El servidor provee el servicio al cliente.

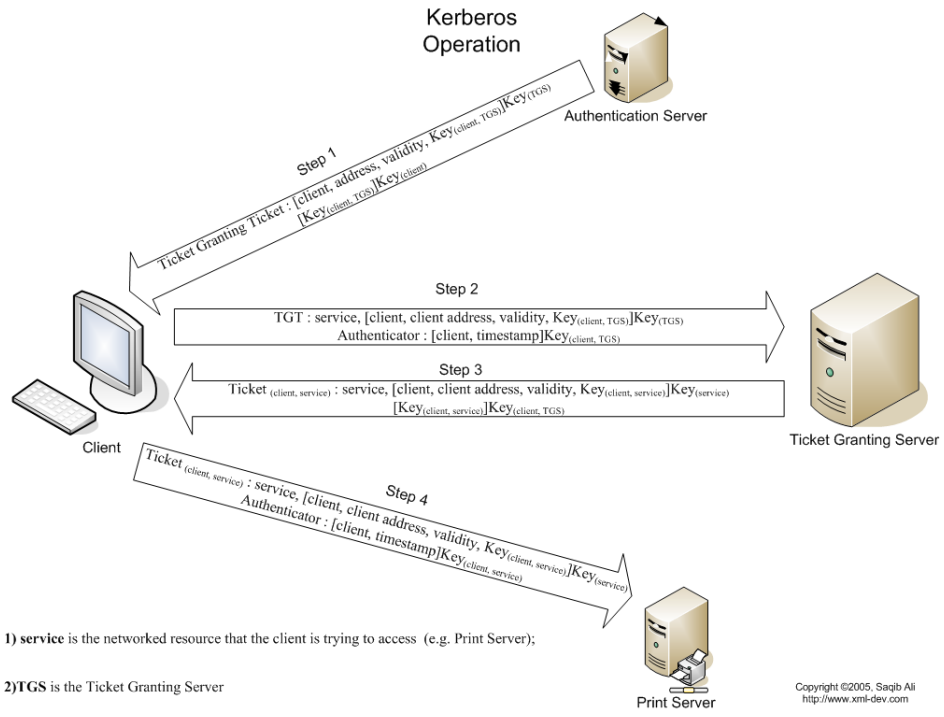

Figura 9: Protocolo Kerberos

4. Control de Acceso

Debido a que a parte de los ataques por red también se pueden dar ataques locales, con tal de preservar las características de confidencialidad, integridad e irrefutabilidad, se necesita un control sobre el acceso a los recursos de cada agente en concreto. En Magentix, un agente se implementa a bajo nivel como un proceso Linux. Si no hay ningún mecanismo de control de acceso, en principio todos los agentes tienen acceso a los mismos recursos locales (por ejemplo ficheros), con lo que se verían comprometidas las características de seguridad que se quieren ofrecer. La solución directa es utilizar los mecanismos de control de acceso que proporciona Linux de forma correcta.

La mayoría de las actuales Plataformas de agentes están desarrolladas en el lenguaje de programación Java y se ejecutan encima de la Java Virtual Machine (JVM). En este tipo de Plataformas un agente normalmente es un hilo de Java de forma que todos los agentes comparten el espacio de memoria del proceso de la JVM. Así pues, el principal problema cuando se quiere controlar el acceso a recursos en este tipo de plataformas es controlar precisamente qué objetos java puede ser accedidos por qué agente.

Las plataformas de agentes que tienen en cuenta aspectos de seguridad solucionan este problema de forma diferente. Por ejemplo, los agentes en la plataforma CAPA no tienen ninguna referencia a objetos de la plataforma o de otros agentes. Por tanto, los ataques basados en obtener referencias a objetos se evitan. SECMAP encapsula objetos que referencian agentes de forma que dichos objetos sólo se pueden acceder por agentes autorizados. En SeMoA, los agentes no pueden compartir clases. Otras plataformas simplemente usan tecnologías Java ya existentes para el control de acceso a objetos, como por ejemplo Jade y Tryllian ADK. El add-on de seguridad para Jade realiza el control de acceso mediante la tecnología Java/JAAS (Java Authentication and Authorization Service). El mecanismo que usa Tryllian ADK es similar al que usa un navegador que está ejecutando un applet de Java.

Como en Magentix cada agente es un proceso Linux, agentes diferentes no comparten memoria, de forma que no se requiere la introducción de otro mecanismo adicional para evitar accesos de agentes al espacio de memoria de otros. La única excepción posible la tendríamos con la tabla GAT. Tal y como se ha explicado en la sección 2.4, esta tabla se mapea como memoria compartida en todos los agentes y en el AMS de un mismo host, pero

solamente el AMS tiene permisos de escritura y Linux asegura esos permisos.

Para controlar el control de acceso a recursos, ya que Magentix está implementado por encima del SO Linux, se pueden utilizar los mecanismos de control de acceso de este último de forma directa. Por tanto, se utilizan los conceptos de usuarios, grupos de usuarios, y listas de control de acceso detallando los permisos para el usuario que es el propietario del recurso, el grupo propietario del recurso y para el resto de usuarios. Como los agentes Magentix se modelan como procesos Linux que posee un usuario Linux, el control de acceso queda asegurado. Además, tenemos que los agentes del mismo usuario van a compartir los mismos recursos, porque no tiene mucho sentido que agentes con el mismo usuario se ataquen unos a otros.

5. Magentix Seguro

En esta sección se detalla la integración que se ha hecho de la suite MIT Kerberos para dotar a Magentix de los aspectos de seguridad requeridos, así como algunas modificaciones a la implementación original de la plataforma para aumentar el grado de seguridad de la misma, aprovechando las herramientas que ofrece cualquier sistema Unix. El resultado ha sido bautizado como Magentix-S (Magentix Seguro).

5.1. Integración de Kerberos en Magentix-S

Un primer concepto interesante que nos encontramos en Kerberos es el de *principal* (o *principal name*). Un *principal* es el nombre único de un usuario o servicio que tiene permitido autenticarse usando Kerberos, y tienen la forma `raiz[/instancia]@REALM`. Para un usuario típico, la raiz es la misma que su ID de login. La instancia es opcional. Si el *principal* tiene instancia, se separa de la raiz con una barra ("/"). Todos los *principal* en un *realm* tienen su propia clave, que se deriva de las contraseñas para el caso de los usuarios y que se asigna de forma aleatoria para los servicios. Un *realm* es un dominio administrativo, una red que usa Kerberos, compuesta por uno o más servidores llamados KDC (Key Distribution Center) y un número potencialmente grande de clientes.

Aunque los *realm* de Kerberos pueden ser cualquier cadena de carácteres ASCII, existe la convención de hacer que coincida con el nombre de dominio en cuestión, en letras mayúsculas. Por ejemplo, los ordenadores en el dominio `example.com` podrían pertenecer al Kerberos *realm* `EXAMPLE.COM`. Como en Magentix los ordenadores no tienen el porque pertenecer a un mismo dominio, y en principio sólo se va a utilizar un *realm* se ha elegido `MAGENTIX`.

En Magentix-S va a existir el concepto de usuario de la plataforma. Estos usuarios se van a corresponder con *principals* de Kerberos, y van a tener la forma `usuario@MAGENTIX`. No hay que confundir estos usuarios de la plataforma con los usuarios de una máquina Unix.

Así pues, cualquier usuario de Magentix-S primero tendrà que hacer login en el sistema de la forma convencional, con lo que necesitará una cuenta válida en el equipo en cuestión. Cuando requiera la utilización de la plata-

forma tendrà que realizar un login en la misma. Para ello se utiliza un nuevo programa introducido en la distribución de Magentix llamado `mgx_login`.

Por ejemplo, tenemos en una máquina Unix que entre sus cuentas de usuario existe el usuario `carlos`. Cuando `carlos` llega enfrente de la máquina hace un login en el sistema y empieza a usarlo. Cuando quiera utilizar Magentix-S para lanzar algún agente primero tendrá que realizar un login en la plataforma con su usuario Kerberos (por ejemplo, `carlitos@MAGENTIX`).

Existen dos tipos de usuarios diferentes en Magentix-S: el administrador y los usuarios convencionales.

5.1.1. Administrador

El administrador de una plataforma Magentix-S tiene las siguientes funciones:

- Crear y Eliminar los usuarios que van a poder lanzar agentes Magentix.
- Crear y Eliminar los servicios de la plataforma.
- Lanzar la plataforma.

Este usuario es por defecto `mgx/admin@MAGENTIX`, que es el que se ha elegido como nombre por defecto para el administrador de la plataforma. Si este nombre no gusta o se quiere cambiar, se puede crear cualquier otro y darle permisos de administrador (siguiendo la guía de administración de Kerberos).

5.1.2. Usuarios Convencionales

Son los usuarios de la plataforma que tienen permiso para poder lanzar agentes en la misma. El administrador por tanto, tiene que crear un *principal* para cada uno de los usuarios que requieran lanzar agentes. En cualquier momento, el administrador puede quitar el derecho a un usuario de lanzar agentes en la plataforma simplemente eliminando su *principal*.

5.2. Modificaciones a Magentix

Para poder cumplir con todos los requisitos de seguridad, además de la integración de Kerberos en Magentix, se han tenido que hacer algunos cambios para garantizar el funcionamiento seguro. Así, uno de los cambios más importantes es que ahora la plataforma ha de ser lanzada por el usuario `root`. Así pues, en cada ordenador perteneciente a la plataforma, tanto el proceso `magentix` como los procesos correspondientes a los servicios (`ams`, `df`, y los nuevos que se vayan desarrollando) se van a ejecutar en modo de superusuario. La explicación a esto se encuentra en el modelo escogido para el lanzamiento de agentes y la política de compartición de recursos del Sistema Operativo entre ellos que se explica más adelante.

5.2.1. Comunicación entre Servicios

El modulo de comunicación entre los servicios de la plataforma ha sido remodelado de forma que ahora los servicios se comunican de forma segura. Para ello el administrador da de alta para cada servicio un *principal* con una clave aleatoria que además almacena en el fichero keytab por defecto `/etc/krb5.keytab`, y que sólo es accesible por el superusuario. Los *princpipals* para los servicios son del tipo `servicio/host@MAGENTIX`.

Por ejemplo, si el host `pc.ejemplo.com` va a formar parte de la plataforma, el administrador va a tener que crear los *principals*: `magentix/pc.ejemplo.com@MAGENTIX`, `ams/pc.ejemplo.com@MAGENTIX` y `df/pc.ejemplo.com@MAGENTIX`.

Cuando un servicio quiere comunicarse con otro servicio establece un contexto de seguridad como cliente con el *principal* del administrador y como servidor el *principal* del servicio destino. De esta forma, se asegura que el servicio con el que está hablando es el que espera y se encuentra en el host esperado. Además, el servicio destino sabe que le está contactando un servicio con la identidad de administrador de la platforma, y por tanto realizará las peticiones que reciba. Con todo esto nos aseguramos que sólo los servicios de la plataforma pueden intercambiar información entre ellos y que ningún proceso maltintencionado podrá mandar comandos a los servicios o podrá observar la información que se intercambian (se cifran las comunicaciones).

Para la programación de este módulo no se ha utilizado directamente la API de Kerberos, sinó que por motivos de portabilidad, mayor abstracción, y más fácil adaptación a nuevas versiones de Kerberos, se ha utilizado la GSS-API (Generic Security Service API Version 2) [28], utilizando Kerberos como mecanismo de bajo nivel. La GSS-API está formada de un conjunto de funciones con las que, primero obtener credenciales válidas para el servicio que hará el papel de cliente, después crear un contexto de seguridad con el servicio destino utilizando dichas credenciales, y finalmente otras funciones que se usan para cifrar/descifrar la información utilizando el contexto adecuado.

La figura 10 muestra el proceso que se sigue para que dos servicios se comuniquen de forma segura.

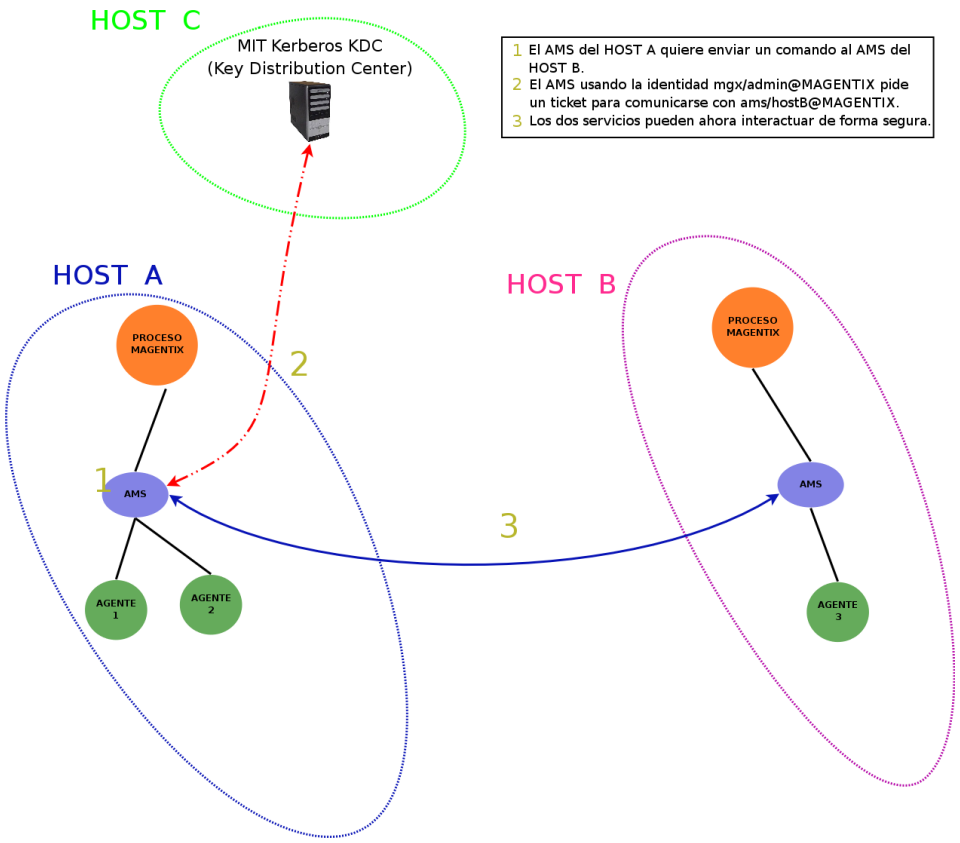

Figura 10: Comunicación segura entre servicios.

5.2.2. Lanzamiento de Agentes

El proceso de lanzamiento de agentes ha sido también objeto de modificaciones para que resulte ser un proceso seguro. En este caso, se quiere asegurar que sólo los usuarios de la plataforma puedan lanzar agentes, y que además éstos se van a lanzar de forma que ningún agente de un usuario pueda tener acceso a recursos del sistema operativo de un agente de otro usuario.

Como primer paso, el usuario tiene que autenticarse contra la plataforma, utilizando el programa `mgx_login` que es un recubrimiento al programa kinit proporcionado en la distribución de Kerberos y que obtiene las credenciales para un usuario.

El segundo paso es lanzar el programa `new_agent`, que ha sido modificado para adaptarlo a las nuevas necesidades. Ahora, además de enviar al AMS el nombre del agente, la ruta del binario y los argumentos, también se envia el `uid` del usuario que ha lanzado el `new_agent` junto con una clave para asegurar que se trata del programa `new_agent`. Además, se cambia el valor de la variable de entorno `KRB5CCNAME` para que apunte al fichero de credenciales del usuario que había realizado el mgx_login, y que se corresponderá con el fichero `/tmp/krb5cc_`*`uid`*. De este modo, al obtener las credenciales desde la GSS-API se obtendrán las del usuario Kerberos correspondiente, ya que como se explica a continuación, el binario `new_agent` estará con el *setuid* activo, y por tanto, se ejecutará con usuario efectivo `root`, cosa que haría que se cogieran por defecto las credenciales del fichero `/tmp/krb5cc_0` que se corresponden con el administrador de la plataforma.

De esta manera, se establece una comunicación segura con el AMS (tal y como se ha explicado en el apartado anterior), pero con la peculiaridad de que para el comando de creación de un nuevo agente el AMS va a permitir que un usuario Kerberos que no sea el administrador le contacte. De esta forma un usuario cualquiera de la plataforma puede pedir al AMS que le lance un agente, pero no podrán pedirle ningún otro comando, puesto que cuando llega cualquier otra petición al AMS sólo aceptará las que provengan con identidad de administrador.

Cuando el comando llega al AMS este, además de lo que hacía con anterioridad, lo procesa de la siguiente manera:

- Comprueba que el comando que le llega es para crear un nuevo agente,

ya que los demás tienen restricciones sobre el usuario que los puede demandar (sólo el administrador).

- Comprueba que la clave que ha recibido es la que él ha creado aleatoriamente en el momento de lanzamiento de la plataforma y que se guarda en el fichero `mgx_file` que sólo es accesible por el usuario `root` y de donde el `new_agent` la ha tenido que leer para demostrar que es la implementación de `new_agent` que se espera, de ahí que haga falta el `setuid` en el binario `new_agent`.

- Al lanzar a ejecutar el binario del agente con sus argumentos, el AMS cambia el `uid` del proceso lanzado (que en principio es root) al `uid` que se le ha pasado y que se corresponderá con el `uid` del usuario que ha lanzado el `new_agent`.

- Finalmente, el AMS utilizando la identidad de administrador, crea un *principal* para el agente creado que coincide con su nombre (`nombre_agente@MAGENTIX`), con una clave aleatoria y guardando dicha clave en un *keytab* independiente para cada agente y que tendrá el nombre `./`*`nombre_agente`*`.kt`, como propietario el `uid` del propietario del agente y sólo permisos de lectura y escritura para el propietario. Por consiguiente, ningún agente podrá utilizar la clave de un agente de otro propietario y hacerse pasar por él.

Además de esto, comentar que al lanzar el binario del agente, el AMS le pasa las variables de entorno que le llegan a él más 3:

- LD_LIBRARY_PATH. Ya se pasaba anteriormente y se utiliza para que se cargue la librería del agente.

- KRB5_KTNAME. Indica la ruta del `keytab` que debe utilizar el `kinit` par buscar la clave necesaria para autenticar al agente.

- KRB5CCNAME. Indica la ruta a utilizar para que el `kinit` una vez autenticado el agente, guarde las credenciales que después se utilizarán desde la GSS-API dentro de los agentes.

La figura 11 muestra un resumen del proceso que se sigue para el lanzamiento de agentes de forma segura.

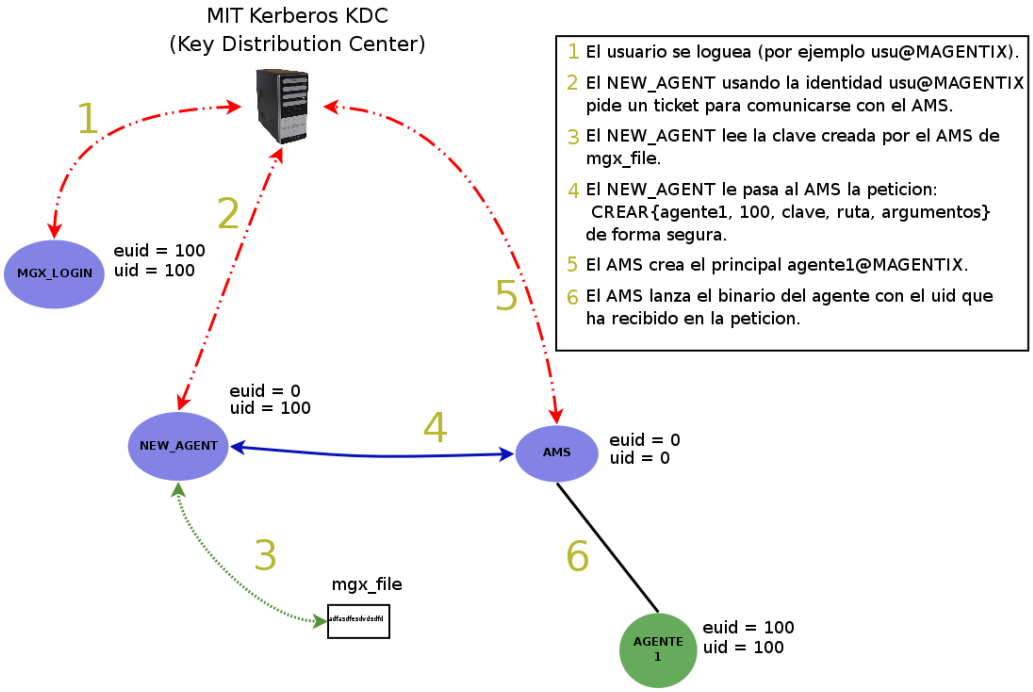

Figura 11: Lanzamiento de agentes.

Por último, cuando el agente acaba su ejecución el AMS destruye su *principal* así como sus ficheros de claves y de credenciales.

5.2.3. Comunicación entre Agentes

Como se ha explicado en el apartado anterior, el AMS crea un *principal* para cada agente nuevo que lanza con lo que modificando el modulo de comunicación entre agentes para que se comuniquen usando un contexto de seguridad entre los dos *principals* que les representan, tendremos que los agentes se comunican de forma segura.

Para la implementación del soporte de Kerberos para este módulo se ha utilizado, al igual que en el caso de los servicios, la GSS-API. Además, también se utiliza como módulo la implementación de `kinit` que viene en la distribución del MIT de Kerberos. Por tanto, además de las funciones que se

hacían anteriormente al inicializar el módulo de comunicación cuando se lanza un agente, se autentica al agente usando el módulo `kinit` que leerá del fichero `keytab` que viene especificado en las variables de entorno y guardará las credenciales obtenidas en el fichero de cache de credenciales que también viene especificado en las variables de entorno.

De esta forma, se obtiene las credenciales del agente ya por medio de la GSS-API, con lo que cuando se crea una conexión nueva con un agente, se crea un contexto nuevo con el agente destino utilizando dichas credenciales. Para un uso eficiente de los contextos, se ha creado una cache de contextos que consiste en una tabla hash de nombre de agente destino y contexto a utilizar. Esta cache es independiente de la cache de conexiones, ya que el número de conexiones está limitado por el número máximo de descriptores de ficheros que linux soporta para un proceso, y en principio, el número de contextos puede ser ilimitado (siempre teniendo en cuenta restricciones de memoria).

Con todo esto tenemos que, aunque se cierre una conexión con un agente, no se pierde el contexto de seguridad asociado. Sin embargo, los contextos no tienen una validez infinita, así que si al intentar utilizar un contexto de seguridad para cifrar o descifrar se descubre que ya no es válido, se negociará uno nuevo.

En la figura 12 se muestra un resumen de los pasos importantes que atañen a la seguridad y se siguen cuando dos agentes se comunican. Hay que dejar claro que el paso 1 sólo se realiza una vez en la vida del agente y es en el momento que el AMS lo crea.

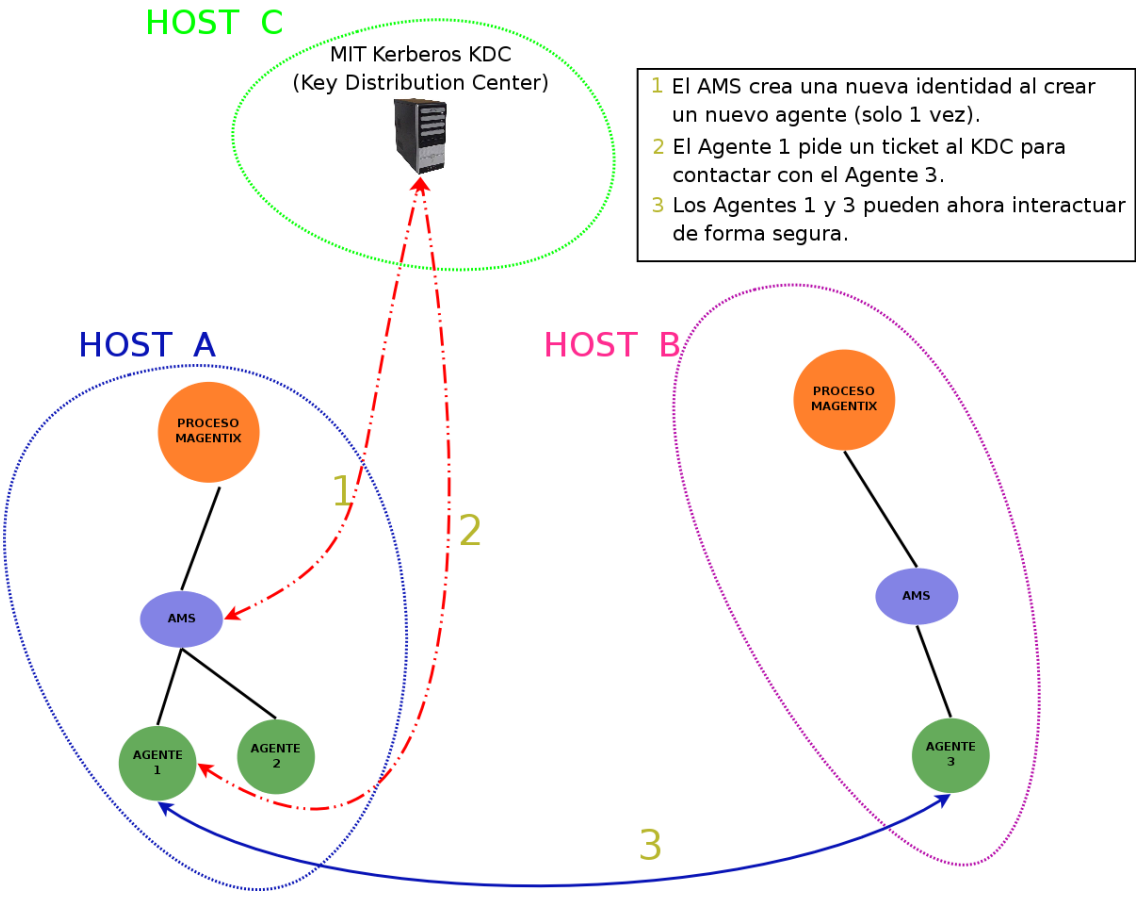

Figura 12: Comunicación entre Agentes

6. Evaluación de la Eficiencia

En esta sección se desarrolla una evaluación de la eficiencia de Magentix versus Magentix-S. Al incluir características de seguridad en Magentix, obviamente la eficiencia se va a ver afectada. Por tanto, es necesario un estudio para cuantificar la pérdida de prestaciones que se tiene al asegurar las interacciones en Magentix, de forma que Magentix pueda continuar siendo una plataforma adecuada para el desarrollo de SMA que tengan ciertos requerimientos en cuanto a eficiencia se refiere.

La comunicación entre agentes con Magentix-S se realiza asegurando la integridad y la confidencialidad de los mensajes intercambiados mediante cómputos criptográficos bastante costosos, de forma que el envío de mensajes se vuelve menos eficiente que si no se encriptan los mensajes. Ya que los SMA son sistemas distribuidos con muchas interacciones, asegurar estas características en los mensajes podría conducir a aplicaciones de SMA que fueran seguras pero que no resultasen mínimamente eficientes.

En las pruebas realizadas, se compara el envío de mensajes entre la versión normal de Magentix y Magentix-S. Con este fin, se ha diseñado un experimento en el que existen parejas de agentes (un agente emisor y un receptor) que se intercambian 1000 mensajes de un tamaño fijo (10 bytes). El tiempo que se recoje es el tiempo desde que la primera pareja empieza a enviarse mensajes hasta que la última termina. El número de parejas existentes en el sistema se incrementa cada vez.

El test que se presenta se ha realizado usando 2 ordenadores Intel(R) Pentium(R) 4 CPU 3.00GHz, con 1GB de RAM memory, y ejecutando el SO Ubuntu Linux 6.06 (kernel 2.6.15). Para la versión segura de Magentix, Kerberos ha sido configurado para usar el algoritmo AES con claves de 128 bits para encriptar y la función hash SHA-1 con claves de 96 bits para las computaciones HMAC.

La figura 13 muestra los resultados obtenidos. Se puede observar claramente que cuando se está usando Magentix seguro hay un sobrecoste tal y como cabía esperar. Sin embargo, este sobrecoste parece ser suficientemente pequeño como para que la versión segura de Magentix permita el desarrollo de SMA seguros con requerimientos de eficiencia.

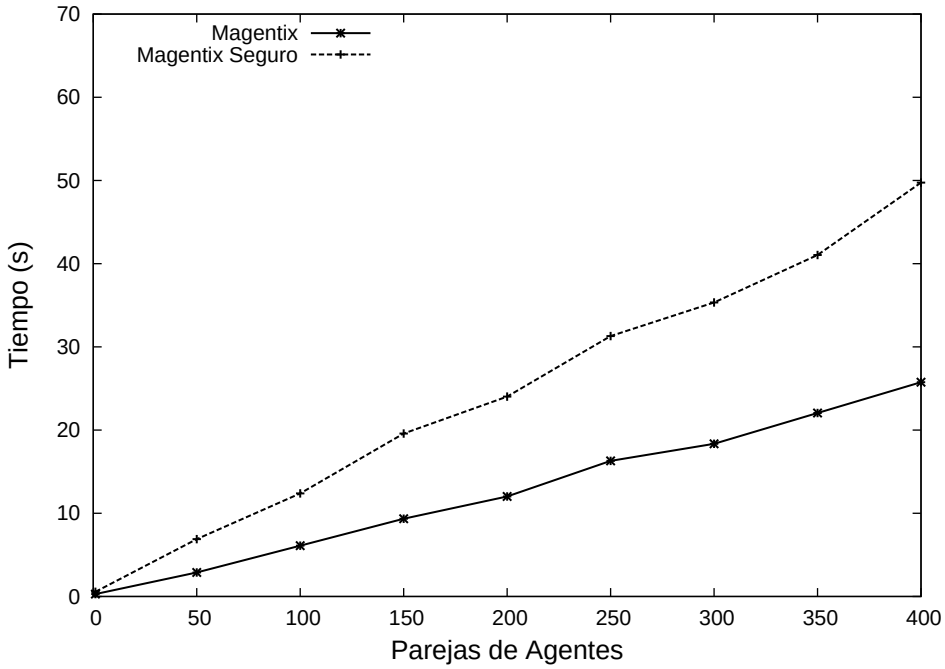

Figura 13: Magentix vs Magentix seguro.

7. Instalación y Configuración de Magentix Seguro

Para una instalación de Magentix Seguro, se tienen que instalar por separado la plataforma Magentix y la distribución de kerberos del MIT.

7.1. Instalación y Configuración de Kerberos

Para instalar la distribución de Kerberos del MIT, el primer paso es acudir a la página web `http://web.mit.edu/Kerberos/` y bajar los fuentes de la última versión disponible (a fecha de la realización de este documento la 1.6.2).

7.1.1. Instalación

Una vez bajado el fichero `krb5-1.6.2-signed.tar` lo descomprimimos y comprobamos la firma con los siguientes comandos:

```
tar xvf krb5-1.6.2-signed.tar
gpg --verify krb5-1.6.2.tar.gz.asc
tar xzvf krb5-1.6.2.tar.gz
```

De esta manera ya tenemos los fuentes disponibles. Seguidamente nos movemos al directorio de las fuentes y configuramos el paquete:

```
cd krb5-1.6.2/src
./configure
```

Por defecto se va a instalar en `/usr/local/`, si queremos cambiar este directorio habrá que mirar la ayuda del configurador ejecutando:

```
configure --help
```

Compilamos el paquete con:

```
make
```

Comprobamos que todo se ha compilado bien con:

```
make check
```

Ahora llega el momento de instalarlo, para ello hay que ser superusuario. En una distribución linux *a lo debian* basta con hacer:

```
sudo make install
```

En este momento la instalación está realida en `/usr/local/`, pero además se tienen que crear algunos directorios adicionales de forma manual:

```
sudo mkdir -p /usr/local/var/krb5kdc
sudo chown root /usr/local/var/krb5kdc
sudo chmod 700 /usr/local/var/krb5kdc
```

7.1.2. Configuración del Realm

Como ya se ha explicado en la sección 5.1, el *realm* que usaremos es `MAGENTIX`. De esta forma se tiene que crear el fichero `/etc/krb5.conf` que va a contener toda la información necesaria sobre ese *realm*. Además, este fichero tiene que estar presente en todos los ordenadores que vayan a formar parte de la plataforma Magentix.

En la figura 14 se muestra un ejemplo de fichero de configuración de Kerberos. En principio y para una instalación por defecto sólo se tendría que cambiar `pc1.example.com` por el ordenador que queramos que contenga el `kdc` y el programa servidor de la administración de Kerberos `kadmind`. Este ordenador puede ser el ordenador que vaya a ser el principal de la plataforma, cualquier otro ordenador que vaya a formar parte de la plataforma, o incluso

un ordenador que no vaya a formar parte de la plataforma. La elección depende del administrador y de la carga que vaya a tener la plataforma. Además, también se va a cambiar en la sección de dominios el dominio `example.com` por el nuestro. Con esto ya tendremos un fichero de configuración de Kerberos listo para usar.

```
[libdefaults]
default_realm = MAGENTIX

# Here, we specify the kdc and admin server for the realm
# LOCAL.NETWORK
[realms]
MAGENTIX = {
  kdc = pc1.example.com
  admin_server = pc1.example.com
}

# This informs the kdc of which hosts it should consider part of the
# LOCAL.NETWORK realm
[domain_realm]
example.com = MAGENTIX
.example.com = MAGENTIX

# I disable kerberos 4 compatibility altogether.  I understand it had
# some real security issues.  I don't know if this is important here,
# but, it doesn't hurt in my particular case (all clients on my network
# are kerberos 5 compatible).
[login]
krb4_convert = false
krb4_get_tickets = false
```

Figura 14: Fichero de configuración de Kerberos (krb5.conf)

Aparte del fichero de configuración de Kerberos en general, se necesita también otro propio del centro de distribución de claves o KDC. Este fichero

ha de estar alojado en `/usr/local/var/krb5kdc/kdc.conf` y en la figura 15 se muestra un ejemplo que se puede utilizar sin modificación.

```
[kdcdefaults]
        kdc_ports = 750,88

[realms]
        MAGENTIX = {
                database_name = /usr/local/var/krb5kdc/principal
                admin_keytab = FILE:/usr/local/var/krb5kdc/kadm5.keytab
                acl_file = /usr/local/var/krb5kdc/kadm5.acl
                key_stash_file = /usr/local/var/krb5kdc/.k5.MAGENTIX
                kdc_ports = 750,88
                max_life = 10h 0m 0s
                max_renewable_life = 7d 0h 0m 0s
        }
```

Figura 15: Fichero de configuración del kdc (kdc.conf)

En el ordenador que vaya a ser nuestro KDC (en el ejemplo anterior `pc1.example.com`), ejecutaremos:

```
sudo /usr/local/sbin/kdb5_util create -s
```

Seguidamente vamos a crear la cuenta de administrador, para ello ejecutaremos:

```
sudo /usr/local/sbin/kadmin.local
```

Dentro de la interfaz que nos ofrece este programa ejecutamos:

```
add_policy -maxlife 180days -minlife 2days -minlength 8 -minclasses 3
           -history 10 default
```

```
addprinc mgx/admin
```

Ahora vamos a darle los permisos necesarios a `mgx/admin@MAGENTIX`, para ello modificamos el fichero `/usr/local/var/krb5kdc/kadm5.acl`, para que todos los administradores tengan acceso a modificar la base de datos de Kerberos, quedando de la siguiente forma el contenido de dicho fichero:

```
*/admin@MAGENTIX *
```

Llegado este punto ya tenemos Kerberos instalado y configurado.

7.2. Sincronización de los ordenadores

Como la seguridad de la autenticación de Kerberos se basa en parte en los time stamps de los tiquets, tener los relojes sincronizados de los ordenadores es crítico, ya que para evitar ataques de fuerza bruta o de repetición se usa un tiempo de vida corto para los tiquets. Si se permite que los relojes de los ordenadores vayan a la deriva, se estará haciendo la red vulnerable a dichos ataques. Como la sincronización de los relojes es tan importante para el protocolo Kerberos, si los relojes de los ordenadores no se sincronizan dentro de una ventana de tiempo razonable, Kerberos mostrará errores fatales y no funcionará. Así, los clientes que intenten autenticarse desde una máquina que tenga un reloj con una diferencia notable sobre el reloj donde se aloja el KDC, verán como sus intentos por autenticarse son siempre fallidos.

Para solucionar esto, se puede utilizar por ejemplo el Network Time Protocol (NTP) que está disponible para sincronización de ordenadores en red y nos va a proporcionar una sincronización sufienctemente precisa como para que el protocolo Kerberos funcione sin problemas. Un gran número de servidores NTP públicos están disponibles para sincronizarse con ellos. NTP es capaz de sincronizar los relojes de los clientes de una red local (LAN) en una ventana de milisegundos, y de una red amplia (WAN) en una ventana de decenas de milisegundos.

Hay paquetes de NTP disponibles para la mayoría de distribuciones Linux, y solamente hay que editar el fichero de configuración (`/etc/ntp.conf`)

para añadir los servidores NTP que se quieren usar para sincronizar los relojes de los ordenadores de la plataforma. Finalmente, se pone una tarea del cron para que se realize la sincronización de forma automática:

```
30 * * * * /usr/sbin/ntpdate -s
```

Si nos encontramos detrás de un cortafuegos los argumentos deberán ser `-su` en vez de `-s` para que `ntpdate` use un puerto sin privilegios para las conexiones de salida hacia los servidores de NTP.

7.3. Instalación de Magentix

Esta es la parte más sencilla de la instalación. Magentix se puede obtener bajando el *source tarball* que se encuentra en la sección *downloads* de la página principal de magentix (`http://www.dsic.upv.es/users/ia/sma/tools/magentix/index.html`). Magentix también se puede acceder via subversion desde el repositorio `https://gti-ia.dsic.upv.es/svn/magentix/linux/trunk`. Puede que no se tenga permiso, así que en ese caso para obtener los fuentes enviar un e-mail a los autores de este texto.

Una vez hemos obtenido los fuentes descomprimiendo el *source tarball* bajado o realizando un `checkout` (ver la ayuda de subversion) realizamos un `make` con lo que ya tendremos los binarios y bibliotecas necesarias para ejecutar Magentix.

7.4. Configuración de Magentix-S

Para configurar Magentix vamos a lanzar los dos programas que van a tener que estar operativos siempre que se esté usando la plataforma y que se corresponden con el centro de distribución de llaves o KDC (`krb5kdc`) y el demonio de administración de Kerberos (`kadmind`). Para lanzarlos ejecutamos:

```
sudo /usr/local/sbin/krb5kdc
sudo /usr/local/sbin/kadmind
```

7.4.1. Creación de Servicios

Se necesita crear los *principal* para los servicios de Magentix que se van a ejecutar en cada host. Para ello, utilizaremos el programa de administración (la parte cliente) que viene con la distribución de Kerberos `kadmin`.

Todo este proceso se tiene que hacer como superusuario (en las distribuciones basadas en debian basta con anyadir al principio de los mandatos que se muestran el comando `sudo`, pero otros sistemas tienen otra forma). Primero hacemos login en Kerberos:

```
sudo kinit mgx/admin
```

Nos pedirá la contraseña y ingresamos la que hayamos definido cuando creamos el *principal* del administrador de Magentix. Seguidamente entramos en el interfaz del cliente de administración de Kerberos con la orden:

```
sudo kadmin
```

Una vez dentro se han de ejecutar los siguientes mandatos:

```
addprinc -randkey magentix/pc1.ejemplo.com
ktadd magentix/pc1.ejemplo.com
addprinc -randkey ams/pc1.ejemplo.com
ktadd ams/pc1.ejemplo.com
addprinc -randkey df/pc1.ejemplo.com
ktadd df/pc1.ejemplo.com
```

Esto habrá que repetirlo para cada uno de los ordenadores que vayan a formar parte de la plataforma, cambiando en cada caso la instancia del *principal*. Por ejemplo, para el caso del servicio AMS en el ordenador `pc2.ejemplo.com` se tendría que dar de alta `ams/pc2.ejemplo.com`.

7.4.2. Creación de Usuarios

Ahora llega el momento de crear los usuarios que van a tener derecho a lanzar agentes en nuestra plataforma. En este momento podemos añadir los que queramos, pero se pueden ir añadiendo después conforme se van necesitando. Primero nos logeamos en Kerberos (si venimos directamente del apartado anterior no hará falta volvernos a logear):

```
sudo kinit mgx/admin
```

Lanzamos el cliente de administración de Kerberos:

```
sudo kadmin
```

Por cada usuario se va a realizar:

```
addprinc usuario
```

Esto creará el *principal* `usuario@MAGENTIX`.

7.5. Lanzamiento de Magentix-S

Una vez instalado y configurado Magentix-S llega el momento de lanzar y disfrutar de la plataforma. Para ello ejecutamos:

```
sudo mgx_login mgx/admin
sudo ./magentix security
```

Con esto habremos lanzado el host principal de magentix, que supongamos que es `pc1.ejemplo.com`. Si ahora queremos añadir un nuevo ordenador a la plataforma, ejecutaríamos en ese ordenador:

```
sudo mgx_login mgx/admin
sudo ./magentix -d pc1.ejemplo.com security
```

7.6. Lanzamiento de Agentes

Como parte final ya de esta guía de instalación, configuración y puesta en marcha de Magentix-S, vamos a explicar como cualquier usuario puede lanzar agentes. De esta forma, cualquier usuario que haya iniciado sesión en alguno de los ordenadores que forman parte de la plataforma y que además disponga de un *principal* de Kerberos podrá lanzar sus agentes. Para ello tendrá que ejecutar los comandos:

```
mgx_login principal_usuario
./new_agent -s nombre_agente ruta_binario argumentos
```

El login de Magentix sólo se tiene que realizar una vez, por tanto, realizadas estas órdenes, si se quieren lanzar más agentes bastará con ejecutar el comando `new_agent`.

8. Conclusiones y Trabajo Futuro

Se ha conseguido dotar a Magentix de un nivel de seguridad que comprende desde autenticación, autorización, integridad, confidencialidad, hasta no repudio en las comunicaciones de los agentes de la plataforma. La versión resultante se ha denominado Magentix-S.

Diferentes alternativas se han tenido en cuenta para dotar a Magentix con las características de seguridad deseadas (autenticación, integridad y confidencialidad). IPsec no resulta válido debido a operar en el nivel de red. Comparando TLS y Kerberos, el último parece resultar más eficiente debido a que únicamente se basa en criptografía simétrica (mucho más eficiente que la criptografia asimétrica) y permite volver a usar contextos de seguridad en diferentes conexiones con el mismo agente (evitando re-negociaciones de contextos de seguridad en cada nueva conexión).

Con esta finalidad se ha integrado la distribución del MIT del protocolo de autenticación de red Kerberos, y además se han hecho mejoras en el diseño de Magentix atendiendo a las posibilidades de seguridad que nos brinda cualquier Sistema Operativo Linux en cuanto a compartición de recursos, y por tanto, impidiendo que agentes de propietarios diferentes puedan interferir entre ellos.

En haras de determinar la degradación de las prestaciones de la plataforma al incluir la infrastructura de seguridad, se ha realizado una evaluación comparando la versión normal de Magentix con la versión segura. Las conclusiones que se obtienen son alentadoras, ya que los resultados obtenidos muestran que el sobrecoste introducido al añadir las características de seguridad deseadas se mantiene en un margen aceptable de forma que la plataforma Magentix se mantiene suficentemente eficiente para permitir el desarrollo de SMA complejos, con un alto número de agentes, con muchas interacciones entre ellos y distribuídos en un gran número de ordenadores en una red muy grande.

Como futuro inmediato queda el poder integrar todo este trabajo de seguridad con otros servicios que se han añadido recientemente en Magentix, como son: el servicio de organizaciones de agentes y el servicio de protocolos de interacción, que dotan a los agentes Magentix de más abstracciones que facilitan su programación y la traducción directa de especificaciones de alto

nivel a agentes Magentix. Por tanto, hay que barajar los nuevos entresijos que se derivan con respecto a la seguridad entre agentes con estos nuevos servicios.

Otra linea futura consiste en aparcar las limitaciones que se tienen cuando se trabaja encima del sistema operativo y que sea el propio sistema el que ofrezca ya las características necesarias. Así, por ejemplo, a la hora de lanzar nuevos agentes en Magentix no haría falta setuidar el proceso `new_agent` para conseguir lo que se quiere, que es que en definitiva el `uid` que llega al AMS sea confiable y que pertenezca al usuario que ha lanzado el `new_agent`. Esto desde dentro del operativo se puede saber enseguida y de forma fiable, por tanto, la plataforma y el sistema operativo donde se lanza alcanzarían un estado de seguridad mayor.

Además, también es necesaria la definición de un esquema general para el control de acceso a los recursos típicos para un SMA y que no son recursos del SO que son los que han tenido en cuenta hasta la fecha. Un ejemplo claro de este tipo de recursos serían los servicios que ofrece cada agente, de forma que un agente pudiese decidir a qué agentes les da acceso a cada uno de los servicios que él publicita. Además, la plataforma Magentix permite la creación de grupos de agentes, por tanto, este esquema general debería permitir especificar qué agentes pueden crear grupos, qué permisos tienen sobre ellos, y qué agentes pueden interactuar con qué agentes del mismo grupo.

Referencias

[1] Fipa (fundation for intelligent physical agents). `http://www.fipa.org`.

[2] Luis Mulet, Jose M. Such, and Juan M. Alberola. Performance evaluation of open-source multiagent 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.

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

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

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

[6] 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*.

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

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

[9] 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.

[10] 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.

[11] Jade. `http://jade.tilab.com`.

[12] S. Ugurlu and N. Erdogan. An overview of secmap secure mobile agent platform. In *Proceedings of Second International Workshop on Safety and Security in Multiagent Systems*, 2005.

[13] H. Xu and S. M. Shatz. Adk: An agent development kit based on a formal design model for multi-agent systems. *Journal of Automated Software Engineering*, 10:337–365, 2003.

[14] M. Duvigneau, D. Moldt, and H. Rolke. Concurrent architecture for a multiagent platform. In *Third International Workshop, Agent-oriented Software Engineering (AOSE)*, 2002.

[15] Cougaar. `http://www.cougaar.org`.

[16] Semoa. `http://www.semoa.org`.

[17] Voyager. `http://www.recursionsw.com/voyager.htm`.

[18] Jose M. Such, Juan M. Alberola, Luis Mulet, Agustin Espinosa, Ana Garcia-Fornes, and Vicent Botti. Large-scale multiagent platform benchmarks. In *LAnguages, methodologies and Development tools for multi-agent systemS (LADS 2007). Proceedings of the Multi-Agent Logics, Languages, and Organisations - Federated Workshops.*, pages 192–204, 2007.

[19] Jose M. Such, Juan M. Alberola, Luis Mulet, Ana Garcia-Fornes, Agustin Espinosa, and Vicent Botti. Hacia el diseño de plataformas multiagente cercanas al sistema operativo. In *International workwhop on practical applications on agents and multi-agent systems*, 2007.

[20] 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 European Workshop on Multi-Agent Systems (EUMAS-2007)*, pages 658–667, 2007.

[21] Juan M. Alberola, Jose M. Such, Vicent Botti, Agustin Espinosa, and Ana Garcia-Fornes. Scalable and efficient multiagent platform close to the operating system. In *Onzè Congrés Internacional de l'Associació Catalana d'Intel·ligència Artificial*, 2008.

[22] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, 1998.

[23] A. Frier, P. Karlton, and P. Kocher. The secure socket layer. Technical Report MSU-CSE-00-2, Netscape Communications, 1996.

[24] T. Dierks and C. Allen. The tls protocol version 1.0. RFC 2246, 1999.

[25] C.Ñeuman, T. Yu, S. Hartman, and K. Raeburn. The kerberos network authentication service (v5). RFC 4120, 2005.

[26] C. Coarfa, P. Druschel, and D. Wallach. Performance analysis of tls web servers, 2002.

[27] Fang-Chun Kuo, Hannes Tschofenig, Fabian Meyer, and Xiaoming Fu. Comparison studies between pre-shared and public key exchange mechanisms for transport layer security. In *INFOCOM*, 2006.

[28] J. Wray. Generic security service api version 2 : C-bindings. RFC 2744, 2000.