UNIVERSIDAD AUTONOMA DE CHIHUAHUA

FACULTAD DE INGENIERIA

Maestría en Ingeniería en Sistemas Computacionales

"MIDDLEWARE BASADO EN AGENTES"

TESINA PARA OBTENER EL GRADO DE MAESTRO EN INGENIERIA
EN
SISTEMAS COMPUTACIONALES

PRESENTA:
I.S.C. Rodrigo Dominguez Garcia

DIRECTOR DE TESINA:
M.C. Pedro Rafael Márquez Gutiérrez

Chihuahua, Chih. Enero del 2010

Dedicatoria…

A mi madre por darme el ejemplo de que todos los objetivos en la vida se logran en base a esfuerzo y dedicación, Y que la obstinación puede ser la mejor virtud sabiéndola aprovechar.

“La formulación de un problema, es más importante que su solución.”

Albert Einstein.

"Si tu intención es describir la verdad, hazlo con sencillez y la elegancia déjasela al sastre."

Albert Einstein.

"Si he visto más lejos es porque he estado parado en los hombros de gigantes."

Isaac Newton

Agradecimientos…

A mi esposa e hija por todo el apoyo y compresión por el tiempo que estuve lejos de ellas durante esta travesía.

A mi asesor por su actitud inquisidora que me alentó a no minimizar mis esfuerzos.

A todos aquellos que dudaron de mi y que colocaron obstáculos en mi camino, por hacer mas interesante y complicado el recorrido de esta travesía.

RESUMEN.

La presente investigación surge como resultado de una serie de problemas identificados durante la administración de una red, como la configuración de acceso de los equipos clientes hacia los servidores, esto consume mucho tiempo dentro de entornos muy dinámicos en donde los cambios realizados en las configuraciones de los servidores impacten en los clientes; otro problema al cual se pretende dar solución es la brecha de seguridad resultante de la distribución de servicios, para este problema el esquema de Middleware brinda una centralización de los accesos, reduciendo la administración al punto de comunicación entre los clientes y los servidores, pero generando un problema aun mas complejo que es la integración de sistemas actualmente en funcionamiento.

Debido a que el esquema de Middleware solo trata la distribución de servicios, se profundiza en el tema de sistema multiagentes para aprovechar la ventajas que nos brinda en la parte de automatización de tareas y envolvimiento de sistemas legados.

Bajo las premisas sobre los beneficios y problemáticas de los esquemas de Middleware y de los sistemas multiagentes, se desarrollo un framework que combina los beneficios de ambos esquemas para dar solución a los problemas previamente identificados.

Índice de contenido

Índice de Imagenes

Capitulo I

Introducción

1.1 Descripción del problema.

Dentro de la administración de un red empresarial existen un sin numero de tareas que requieren una gran inversión de tiempo, por lo cual muchas veces los encargados de estas tareas son fácilmente sobre pasados por las mismas. Uno de los principales objetivos de esta investigación es el desarrollo de una aplicación que permita la automatización de tareas independientemente del sistema operativo de los equipos clientes, además de contar con un esquema centralizado y auto configurable de acceso a los servicios.

En base a los problemas que se desean atender descritos anteriormente surgen las siguientes preguntas

¿Que se requiere para implementar una aplicación de este tipo?

¿Que tan factible es implementarlo?

¿Que tecnologías actuales pueden dar solución a este problema?

1.2 Objetivos.

Para obtener una mayor comprensión del funcionamiento de los servicios dentro de una red y las modificaciones requeridas para la exitosa implementación de la solución propuesta en esta investigación, se llevara acabo un análisis de algunos de los esquemas mas utilizados para la interconexión de aplicaciones como los son Cliente/Servidor y ciertos tipos de Middleware.

Por otro lado la problemática de la automatización de tareas se abordara a través del uso de agentes, lo cual implica hacer una introspección a la teoría de los mismos, así como la evaluación de un framework, para determinar las ventajas y desventajas inherentes a la implementación de aplicaciones mediante esta metodología.

Se realizara un estudio de algunas herramientas para la construcción de sistemas distribuidos, se evaluara si facilitan la implementación de la solución propuesta, se

llevara acabo el desarrollo de un framework tomado las fortalezas de cada una de ellas, pero con el objetivo principal de que facilite el desarrollo e integración de aplicaciones a través de el. Para demostrar la viabilidad del uso de agentes mediante el framework desarrollado, se buscara implementar un sistema que controle el acceso a los servicios dentro de la red.

1.3. Restricciones.

Debido a la extensión del problema la investigación sobre framework para desarrollo de Middleware y aplicaciones multiagentes se centrara en el estudio de RMI y JADE como base para el desarrollo de nuevo framework; así como no se implementaran todas la funciones requeridas por el framework, solo aquellas que sean mas representativas.

1.4. Justificación.

La investigación busca brindar un esquema que simplifique los tiempos de administración de una red, atacando las tareas mas demandantes de tiempo, como lo es la configuración de los equipos cliente.

Otro apunto a considerar es el incremento de la seguridad que se busca, mediante la implementación de un esquema de acceso centralizado el cual sea controlado por agentes, lo cual facilitara en gran medida las tareas administrativas en el marco de seguridad.

Por ultimo el Framework a desarrollar busca brindar un conjunto de herramientas que facilite el desarrollo de aplicaciones usando agentes, explotando las ventajas que estos nos brinda.

1.5. Desarrollo del Documento.

A grandes rasgos la organización del presente documento es como se menciona a

continuación:

Capitulo 2: Ese capitulo se encuentra dividido en 2 partes, la primera expone los conceptos sobre sistemas distribuidos y sistemas multiagentes, y la segunda hace una revisión a 2 importantes aplicaciones, RMI como ejemplo de un Middleware y JADE para la construcción de sistemas multiagentes.

Capitulo 3: describe las ideas bases para el diseño del framework, además se explican diversos diagramas UML utilizados para modelarlo; Terminando con una explicación de cada uno de los Componentes que forman parte del framework.

Capitulo 4: Consta de 2 partes, la primera realiza una demostración del funcionamiento del framework a través de la implementación de una serie de ejemplos que demuestran las funcionalidades básicas del mismo; y la segunda parte demuestra la implementación de un Middleware básico para controlar los accesos a los servicios dentro de una red controlado por los agentes del sistema.

Capitulo 5: Se contestan a las preguntas realizadas al inicio de la investigación, así como el planteamiento de futuras mejoras y áreas de oportunidad para el uso del framework desarrollado.

Capitulo II

Análisis de Fundamentos

2.1 Arquitectura Cliente-Servidor.

Este modelo [1-4,6] se sustenta en la idea de distribuir el procesamiento y almacenamiento de información, en equipos de mayor potencia denominados servidores, permitiendo que equipos de poco o nulos recursos puedan utilizar estos recursos dentro la red, además de brindar un esquema de seguridad centralizado de restricción de acceso.

Durante la implementación del modelo cliente-servidor, se deben definir las funcionalidades que tendrá el servidor y las funcionalidades del cliente, ya que toda aplicación informática consta de tres niveles, los cuales son:

- Nivel de Presentación: conformada por la interfaz de usuario, dentro de la cual se realiza la captura y despliegue de los datos del sistema.
- Nivel de Lógica de Negocio: Son los procedimientos que definen el manejo de la información dentro del sistema.
- Nivel de Acceso a Datos: consta del almacenamiento físico de los datos y de la extracción, consulta y actualización de los datos.

En base a esta división de funcionalidades existen varios esquemas de funcionamiento para este modelo [7].

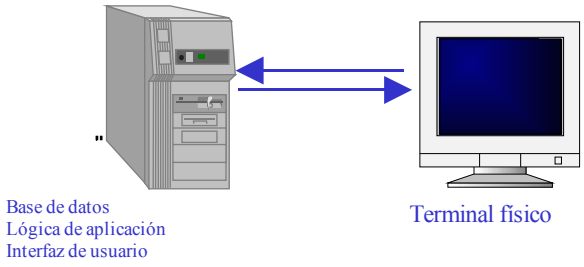

Fig. 2.1 Esquema de representación distribuida.

Representación distribuida.- El cliente se conecta mediante la red al servidor, entablando una sesión mediante la cual interactúa con la aplicación residente en el servidor, en este esquema todos los procesos de la aplicación están corriendo del lado del servidor.

Representación remota.- En el lado del cliente se ejecuta la interfaz básica de usuario, la cual formatea los datos enviados por el servidor y realizar las acciones de interacción con el usuario; la lógica de la aplicación y la base de datos radican en el servidor

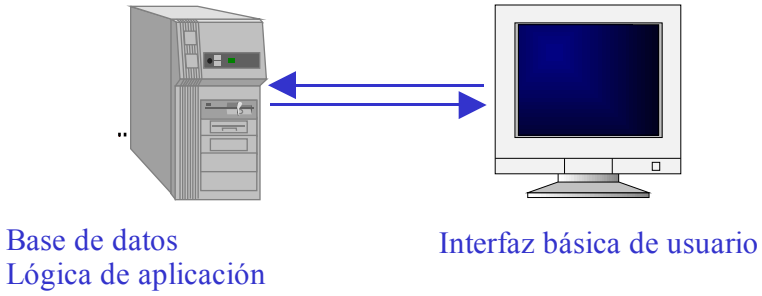

Fig. 2.2 Esquema de representación remota.

Lógica distribuida.- En el cliente además de la interfaz de usuario se realizan validaciones básicas de la aplicación, como formato de ingreso de los datos, datos obligatorios, etc.; en el lado del servidor se mantiene la lógica de procesos de la aplicación y la base de datos.

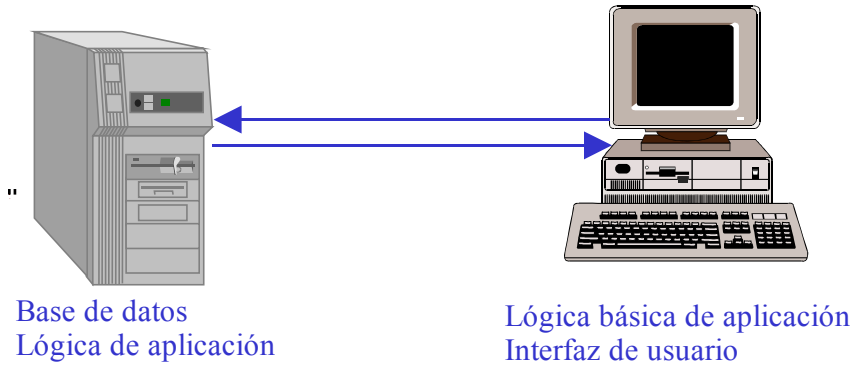

Fig. 2.3 Esquema de lógica distribuida.

Gestión remota de datos.- La interfaz de usuario y la lógica de la aplicación radican en el cliente, solo la base de datos se encuentra en el servidor.

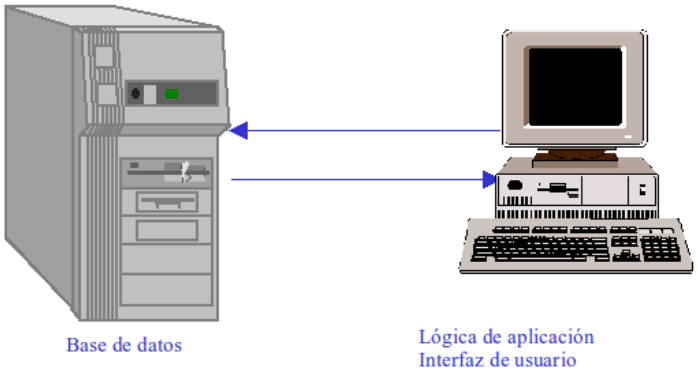

Fig. 2.4 Esquema de gestión remota.

Ventajas [3,5].

- Proporcionan un mejor acceso a los datos. La interfaz de usuario ofrece una forma homogénea de ver el sistema, independientemente de los cambios o actualizaciones que se produzcan en él y de la ubicación de la información.
- Mejora en el rendimiento de la red, reduciendo la necesidad de mover grandes cantidades de información por la red hacia las estaciones de trabajo para su procesamiento. Esa tarea la controlan los servidores, procesando las peticiones y después transfiriendo sólo los datos requeridos a la máquina cliente.
- Permite centralizar el control de acceso a los recursos.

Desventajas [3,5].

- Requiere un fuerte rediseño de todos los elementos involucrados en los sistemas de información (modelos de datos, procesos, interfaces, comunicaciones, almacenamiento de datos, etc.).
- Es más difícil asegurar la de seguridad de los datos ya que el equipo es accesible a través de la red.
- A veces, los problemas de congestión de la red pueden degradar el rendimiento del sistema.

2.2 Middleware.

Es un software de computadora [8-10] que consiste en un conjunto de servicios que permiten que varios procesos que corren sobre diferentes equipos se comuniquen a través de al red, para que puedan intercambiar datos entre plataformas heterogéneas. Utilizado para soportar aplicaciones distribuidas, facilitar conexión a sistemas legados o basados en protocolos de comunicación propietarios, pero esencialmente para implementación de arquitecturas orientada a servicios (SOA).

El Middleware proporciona un conjunto de interfaces que permiten a las aplicaciones:

- Transparencia de localización dentro de la red.
- Independencia de los protocolos de comunicación.
- Facilitar escalabilidad de servicios dentro de la red

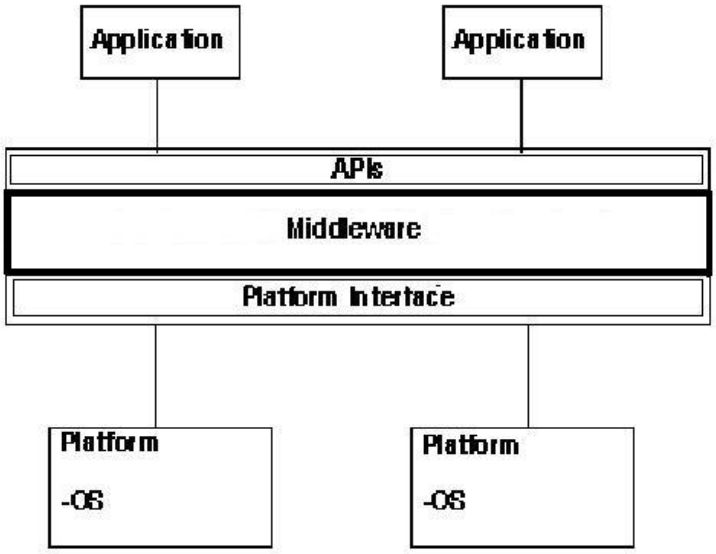

Fig. 2.5 Arquitectura de Middleware.

2.2.1 Tipos de Middleware.

Un ejemplo de los diversos esquemas de funcionamiento de un Middleware son [8-10]:

- Llamadas a procedimientos remotos (RPC).
- Object request broker (ORB).
- Middleware Orientado a mensajes (MOM).

2.2.2 Llamadas a procedimientos remotos (RPC).

Es un protocolo de comunicación mediante el cual una aplicación cliente solicita a otro equipo dentro de la red que ejecute un procedimiento y le regrese un resultado. Las llamadas a procesos remotos no son de forma concurrente, si no que el equipo que manda llamar el procedimiento remoto se queda bloqueado en espera del resultado de salida del mismo [11].

La invocación a un procedimiento remoto se realiza cuando el cliente hace la solicitud pasando los parámetros de entrada mediante un cliente-stub al servidor, el cual mediante el Server-stub define el procedimiento que ha sido invocado, realizando las operaciones asociadas al mismo y regresando el valor de salida, desbloqueándose el equipo cliente y así continuar con su ejecución normal.

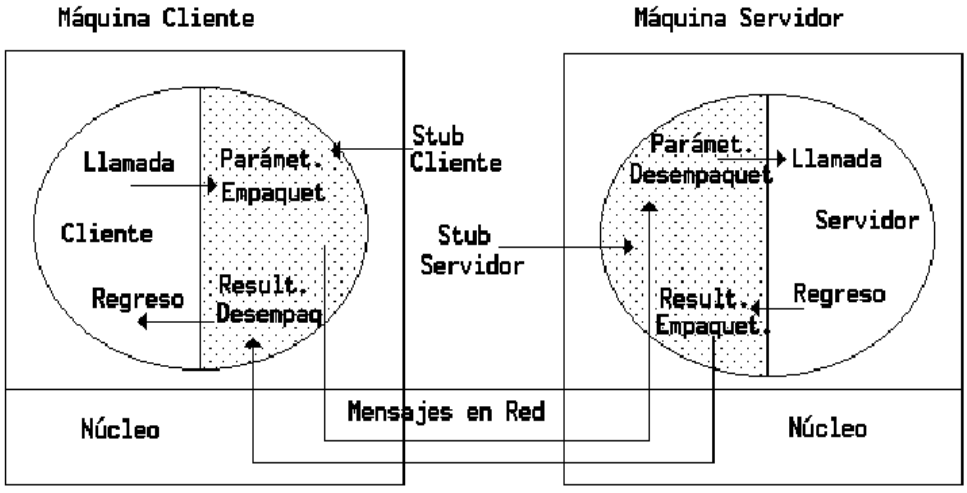

Fig. 2.6 Funcionamiento de una llamada a un procedimiento remoto.

2.2.3 Object Request Broker (ORB).

Define una interfaz de comunicación entre aplicaciones semejante al mecanismo de RPC, con la diferencia de que ejecuta un objeto de manera remota, en lugar de un procedimiento. Brindando un servicio de directorio mediante el cual las aplicaciones clientes localizan y utilizan los servicios que brindan los objetos remotos [12].

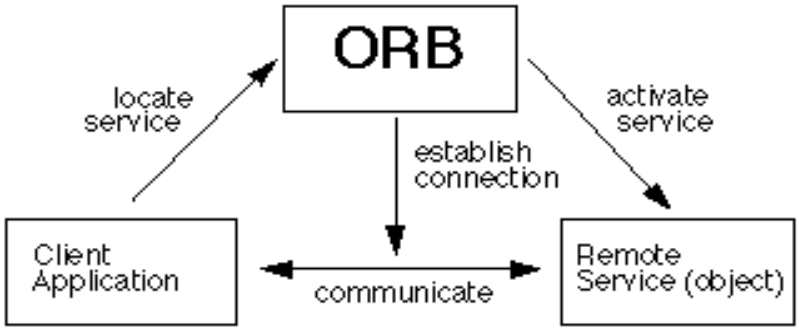

Fig. 2.7 Esquema de funcionamiento de utilización del esquema de objetos remotos.

2.2.4 Middleware orientado a mensajes (MOM).

Es el software que reside en ambas porciones de una arquitectura cliente/servidor y apoya principalmente llamadas asincrónicas entre el cliente y los servidores de aplicaciones. Las colas de mensaje proporcionan el almacenamiento temporal cuando el programa de destino está ocupado o no está conectado.

Los mecanismos de comunicación asíncronos y síncronos, tienen fuerzas y debilidades que deben considerarse al momento de realizar el diseño de una aplicación. MOM proporciona los dos métodos de comunicación, donde el comportamiento del mecanismo síncrono es semejante al del RPC; el equipo cliente envía un mensaje y se queda bloqueado en espera de la respuesta. El uso de las colas de mensaje, tiende a ser más flexible que los sistemas basados en RPC, ya que pueden omitir el mecanismo síncrono y convertirse en asíncrono, si un servidor llega a ser inaccesible. Además permite realizar

muchas respuestas a una petición o muchas peticiones a una respuesta, a continuación se muestra el esquema de funcionamiento del MOM [12].

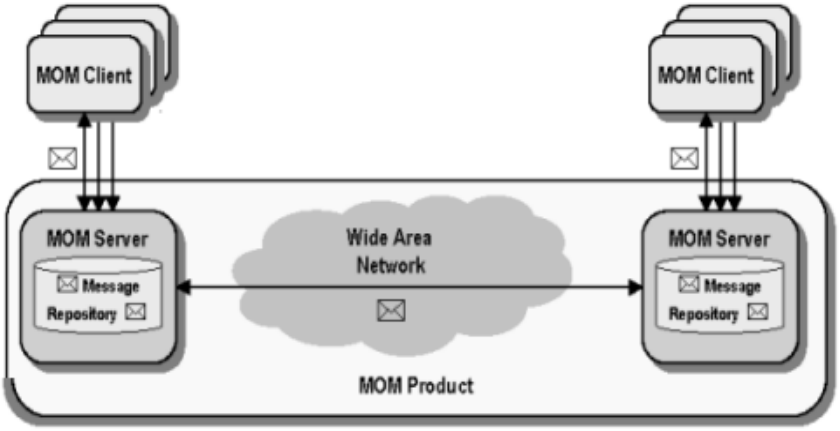

Fig. 2.8 Esquema de funcionamiento de un Middleware orientado a mensajes.

Modelos de funcionamientos.

Existen 2 tipos de modelos de funcionamiento para el envío de mensajes.

- Point to point.
- publish-subscribe.

Modelo Point to Point.

Los sistemas Point to Point trabajan con colas de mensajes. En los sistemas Point to Point, los mensajes se encaminan a un consumidor individual que mantenga una cola de mensajes entrantes. Las aplicaciones de mensajerías envían mensajes a una cola específica y los clientes extraen mensajes de esa cola.

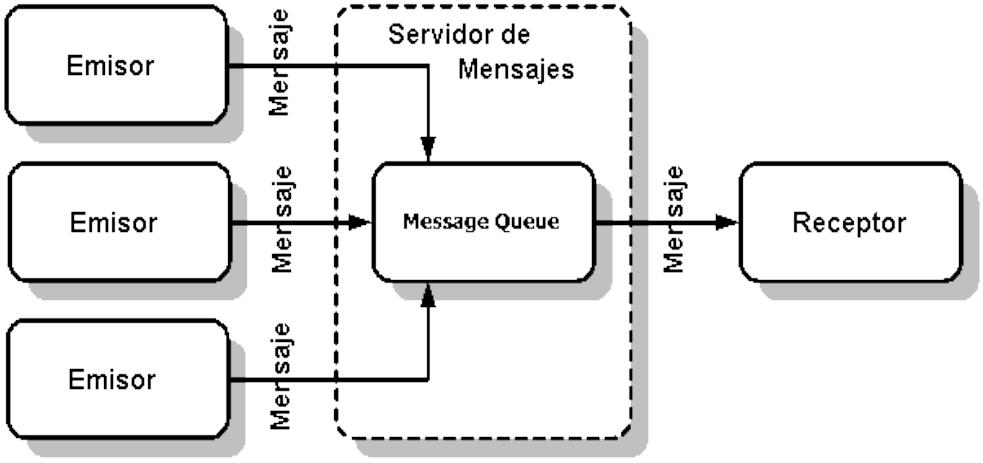

Fig. 2.9 Modelo point to point.

Modelo Publish-Subscribe.

Se usa este modelo de mensajería cuando múltiples aplicaciones quieren recibir el mismo mensaje a la vez. Un ejemplo de este modelo puede ser un Chat. Es muy útil cuando un grupo de aplicaciones quieren notificar a cada una de las otras aplicaciones una ocurrencia en particular. En este caso se presenta un esquema de comunicación de orden M-N.

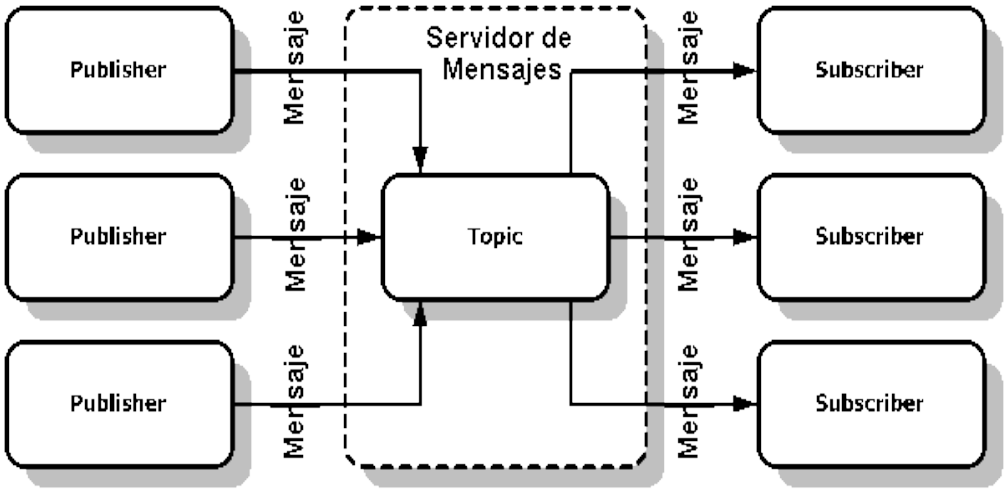

Fig. 2.10 Modelo Publish-Subscribe.

2.3 Metodología de agentes.

La programación orientada a objetos representa un mayor nivel de abstracción que la programación basada en procedimientos; es más fácil de comprender y de mantener, por lo tanto, más productiva. La metodología de agentes, establecen una serie de mecanismos que pretenden dar un paso más allá en el tratamiento informático distribuido, permitiendo la interacción dinámica de componentes autónomos y heterogéneos [13]. Los agentes móviles añaden una singularidad especial al concepto de agentes: la posibilidad de trasladarse de una máquina a otra. Esta característica ofrece ciertas ventajas respecto al tratamiento de la información en modo cliente/servidor, sobre todo en la computación a través de Internet. El auge mundial de "la red de redes" ha permitido el rápido desarrollo de nuevas técnicas inteligentes para búsqueda, filtrado y gestión de datos [13].

2.3.1 Definición de agentes.

Un agente es un sistema informático, situado en algún entorno, que percibe el entorno y a partir de tales percepciones determina y ejecuta acciones de forma autónoma y flexible que le permiten alcanzar sus objetivos y que pueden cambiar el entorno, a continuación e en listan las características de un agente [13]:

- Autónomo.
- Situado en un ambiente.
- Debe reaccionar a los cambios.
- Debe buscar lograr sus objetivos a cualquier costo.
- Debe ser flexible en la forma de conseguir sus objetivos.
- Debe ser robusto en la recuperación de errores.
- Debe interactuar con los demás agentes del medio.

2.3.2 Sistemas Multi agentes.

Normalmente los agentes no actúan en solitario, sino que se localizan en entornos o plataformas con varios agentes, donde cada uno de ellos tiene sus propios objetivos, toma

sus propias decisiones y puede tener la capacidad de comunicarse con otros agentes; dichos entornos se conocen con el nombre de sistemas multiagentes o agencias.

Los sistemas multiagentes dan un mayor nivel de abstracción con respecto a la informática distribuida tradicional, estando más cercanos a las expectativas del usuario y permitiendo al programador una mayor flexibilidad para expresar el comportamiento de los agentes. Como características principales de las agencias podemos destacar las siguientes [13]:

- Cada agente del sistema tiene un punto de vista limitado (no tiene la información completa).
- No existe un control global para todo el sistema.
- Los datos están descentralizados.
- La computación es asíncrona.
- Permite ínter operación con sistemas existentes.

Sin embargo, este tipo de diseños suele plantear una serie de problemas, como la elección del tipo de comunicación entre los agentes, del tipo de plataforma o del método de desarrollo, de los criterios para la seguridad del sistema, etc.

2.3.3 Agentes móviles.

Para comenzar con los agentes móviles, veremos las diferencias principales entre los conceptos de agentes estáticos y agentes móviles.

Agente estático.- Sólo puede ejecutarse en la máquina donde fue iniciado. Si éste necesita interactuar con otros agentes o programas o requiere cierta información que no se encuentra en el sistema, la comunicación puede llevarse a cabo mediante cualquier método de interacción para objetos distribuido, como CORBA o RMI de Java.

Agente móvil.- No está limitado al sistema donde se inició su ejecución, siendo capaz de transportarse de una máquina a otra a través de la red. Esta posibilidad le permite interactuar con el objeto deseado de forma directa sobre el sistema de agentes donde se encuentre dicho objeto. También puede utilizar los servicios ofrecidos por el sistema multiagente destinatario.

Las tareas de búsqueda y tratamiento de la información en Internet tienen últimamente una gran importancia en el desarrollo de sistemas basados en agentes móviles. Debido al rápido crecimiento de la Red, el proceso de encontrar los datos más convenientes para un usuario resulta excesivamente tedioso y complejo. Mediante este esquema, se puede enviar un agente a los destinos más interesantes para el usuario, localizar y filtrar la información deseada siguiendo las normas dictadas por éste y traerla consigo al ordenador de origen, permitiendo ahorrar tiempo de conexión, ancho de banda y brindar una total transparencia de localización de los repositorios de datos al usuario [13].

2.4 RMI.

Java RMI (Invocación Remota de Métodos) es un modelo de objetos distribuidos para la plataforma Java. La principal característica de RMI es que es un modelo cuyo eje central es el lenguaje Java que se enfoca en la homogeneidad del entorno para distribuir de forma más sencilla y eficiente la aplicación basándose en arquitectura virtual común para todo tipo de plataformas, asegurando una gran interoperabilidad.

La idea de Java RMI es la misma que la de CORBA: extender el modelo de invocación de métodos dentro del espacio de direccionamiento de una sola máquina virtual a invocaciones remotas de forma lo más transparente posible para el desarrollador. La invocación de métodos se puede realizar entre diferentes VM (Virtual Machine) interconectadas a través de una red de comunicación.

Una característica a destacar de Java RMI es que incluso los objetos pueden pasarse como parámetros de una invocación, o como valores de retorno. Para ello SUN ha desarrollado una metodología para transformar los objetos en flujos de octetos (byte-stream) y así poder ser enviados por los canales de comunicación. A esta metodología se la conoce como serialización de los objetos (object serialization). Es una característica que también es útil para almacenar en algún medio persistente a los objetos, algo fundamental por ejemplo en las bases de datos de objetos.

En SUN ven a Java RMI como una extensión natural al paradigma de los objetos de su RPC (Llamadas remotas a procedimientos). Con esto se logra el objetivo de hacer lo más transparente posible el hecho de que la invocación sea remota, de forma que los programadores tienen un modelo común de objetos tanto para aplicaciones centralizadas como distribuidas, el modelo de objetos de Java.

A la sencillez que se logra al centrar todo el modelo entorno a un lenguaje común se une el hecho de que la máquina virtual Java tiene un recolector de basura automático, por lo que la gestión de la memoria distribuida, uno de los temas más espinosos dentro de cualquier sistema de objetos distribuidos, es transparente para el programador [14-18].

2.4.1 Arquitectura RMI.

La arquitectura RMI puede verse como un modelo de cuatro capas [16,17]:

- Capa de aplicación.
- Capa de Proxy o capa stub/skeleton.
- Capa de referencia remota.
- Capa de transporte.

Cada capa es independiente de las otras y tiene definida su propia interfaz y protocolo, de forma que una capa puede ser cambiada sin afectar a las otras. Por ejemplo, la capa de transporte puede utilizar distintos protocolos como TCP, UDP, etc... sin que el resto de las capas se vean afectadas en su funcionamiento.

Capa de aplicación.- Corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda java.rmi.Remote. Dicha interfaz se usa básicamente para "marcar" un objeto como remotamente accesible.

Capa de Proxy o capa Stub-Skeleton.- Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos se realizan en esta capa, mediante la implementación de stub local del lado del cliente, el cual implementa la interfaz del objeto remoto y gestiona toda la comunicación con el server a través de la capa de referencia remota. En lado del servidor, el Skeleton es el equivalente al Stub en el cliente, el cual se encarga de traducir las invocaciones que provienen de la capa de referencia remota, así como de gestionar las respuestas.

Capa de referencia remota.- Descansa debajo de la capa Stub/Skeleton. Está formada por dos entidades distintas, el cliente y el servidor, que se comunican a través de la capa de transporte. Es responsable del manejo de la parte semántica de las invocaciones remotas, gestión de la replicación de objetos e implementación del protocolo de comunicación, que puede ser de distintos tipos:

- Invocación unicast punto-punto.
- Invocación a grupos de objetos.
- Estrategias de reconexión.

Capa de transporte.- Responsable del establecimiento y mantenimiento de la conexión, proporcionando una canal de comunicación fiable entre las capas de referencia remota del cliente y del servidor. Sus principales responsabilidades son:

- Establecimiento y mantenimiento de la conexión.
- Atender a llamadas entrantes.
- Establecer la comunicación para las llamadas entrantes.

2.4.2 Stubs y Skeletons

RMI utiliza el mismo mecanismo que los sistemas basados en RPC para realizar la comunicación con los objetos remotos, utilizando definición de Stubs y Skeletons. Los "Stubs" implementan el mismo conjunto de interfaces remotas que el objeto remoto al cual representa, actuando como referencia de los objetos remotos ante sus clientes. En el cliente se invocan los métodos del Stubs, quien es el responsable de invocar de manera remota al código que implementa al objeto remoto.

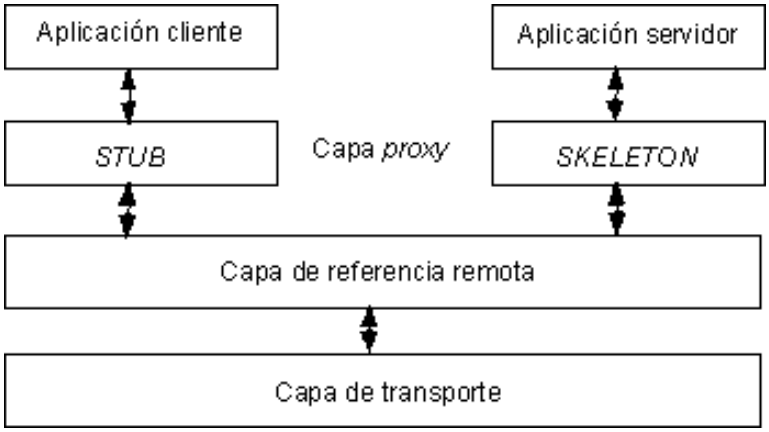

Fig. 2.11 Arquitectura RMI.

Durante la invocación de método remoto, del lado del cliente se realizan las siguientes acciones:

- Inicia una conexión con la VM que contiene al objeto remoto.
- Serializa el objeto y transmite los parámetros de la invocación a la VM remota.

- Espera por el resultado de la invocación.
- Deserializa el objeto y devuelve el valor de retorno o la excepción.
- Devuelve el valor a quien lo llamó.

Los Stubs ocultan la serialización de los parámetros, así como los mecanismos de comunicación empleados. En la VM remota, cada objeto debe poseer un Skeleton correspondiente (en versiones posteriores al JDK 1.2, los Skeletons no son necesarios). El Skeleton es responsable de despachar la invocación al objeto remoto.

Cuando un Skeleton recibe una invocación, realiza las siguientes acciones:

- Deserializa los parámetros necesarios para la ejecución del método remoto.
- Invoca el método de la implantación del objeto remoto.
- Serializa los resultados y los envía de vuelta al cliente.

A partir del JDK 1.2 se agregó un protocolo adicional a los Stubs, para eliminar la necesidad de los Skeletons en el lado del servidor, y en lugar de ellos, se utiliza código genérico para realizar todas las operaciones que eran responsabilidad de los Skeletons en el JDK 1.1 [16].

2.4.3 Implementación de Un Objeto Remoto.

Para mostrar el funcionamiento de RMI. Se presenta una clase que permite realizar la suma de 2 enteros de manera remota. Las clases necesarias son las siguientes:

- InterfaceRemota: Es una interfaz java con todos los métodos que queramos poder invocar de forma remota, es decir, los métodos que queremos llamar desde el cliente, pero que se ejecutarán en el servidor.
- ObjetoRemoto: Es una clase con la implementación de los métodos de InterfaceRemota: A esta clase sólo la ve el servidor de RMI.
- ObjetoRemoto_Stub: Es una clase que implementa InterfaceRemota, pero cada

método se encarga de hacer una llamada a través de red al ObjetoRemoto del servidor, esperar el resultado y devolverlo. Esta clase es la que ve el cliente y no necesitamos codificarla, java lo hace automáticamente para nosotros a partir de ObjetoRemoto.

2.4.4. InterfaceRemota.

Primero se debe definir una interfaz con los métodos que serán llamados de manera remota. Esta interfaz esta definida como sigue:

```
import java.rmi.Remote;
public interface InterfaceRemota extends Remote
{
      public int suma (int a, int b) throws
      java.rmi.RemoteException;
}
```

La clase InterfaceRemota hereda de la interfaz java.rmi.Remote para que el objeto pueda invocar métodos remotos. Además de definir los métodos a invocar, los cuales deben lanzar una excepción java.rmi.RemoteException, que se producirá si hay algún problema con la comunicación entre los dos ordenadores o cualquier otro problema interno de RMI o en el servidor.

Todos los parámetros y valores devueltos de estos métodos deben ser tipos primitivos de java o bien clases que implementen la interfaz java.io.Serializable de java. De esta forma, tanto los parámetros como el resultado podrán ser enviados por la red.

2.4.5 Objeto Remoto.

Se debe desarrollar una clase que implemente los métodos contenidos dentro de la clase InterfaceRemota, que se vayan a invocar desde el cliente RMI. El servidor de RMI se

encargará de instanciar esta clase y de ponerla a disposición de los clientes.

La clase ObjetoRemoto debe heredar java.rmi.UnicastRemoteObject, además implementar los métodos definidos dentro de la clase interfaz InterfaceRemota, la definición de la clase se muestra a continuación:

```
import java.io.Serializable;
public class ObjetoRemoto extends UnicastRemoteObject implements
InterfaceRemota
{
      public int suma(int a, int b)
      {
            System.out.println ("sumando " + a + " + " + b +
            "...");
            return a+b;
      }
}
```

Otra opción es no hacerlo heredar de java.rmi.UnicastRemoteObject, pero luego la forma de registrarlo varía un poco y además debemos encargarnos de implementar adecuadamente todos los métodos requeridos para que funcione de manera correctamente.

2.4.6 ObjetoRemoto_Stub.

Una vez compilado y que obtenemos el archivo ObjetoRemoto.class, necesitamos crear el Stub de la clase. Que nos permite que un programa en una PC pueda llamar a un método de una clase que está en otra, el Stub es una clase con los mismos métodos de la clase ObjetoRemoto, pero en cada uno de los métodos está codificado el procedimiento del envió de mensajes por la red y recepción de la respuesta. Esta codificación se lleva acabo mediante la herramienta rmic, a la cual se le pasa la clase ObjetoRemoto y nos devuelve la Stub ObjetoRemoto_stub.class.

La ejecución de la herramienta rmic es la siguiente:

```
$ set CLASSPATH=”ruta donde se encuentra las clases”
$ rmic ObjetoRemoto
```

Esto generará un ObjetoRemoto_stub.class; en versiones antiguas de java también genera un ObjetoRemoto_Skel.class. El primero debe estar visible tanto por el cliente como por el servidor, es decir, deben aparecer en el CLASSPATH de ambos. Eso implica que debe estar situado en el servidor en un sitio público al que el cliente tenga acceso o que se debe suministrar una copia al cliente. El ObjetoRemoto_Skel.class se genera por defecto y sólo es útil para clientes con java anterior a la versión 1.2.

2.4.7 Puesta en Marcha.

En el PC servidor de RMI deben correr dos programas:

- Rmiregistry: Este programa lo proporciona java (localizado en JAVA_HOME/bin/, donde JAVA_HOME es el directorio de instalación de java). Una vez arrancado, admite que registremos en él objetos para que puedan ser invocados remotamente y admite peticiones de clientes para ejecutar métodos de estos objetos.
- Clase Servidor: Es un clase que instancia el ObjetoRemoto y lo registra en el rmiregistry. Una vez registrado el ObjetoRemoto, el servidor no muere, sino que queda vivo. Cuando un cliente llame a un método de ObjetoRemoto, el código de ese método se ejecutará en este proceso.

En el PC del cliente debe correr el programa:

- Clase Cliente: Este clase pide al rmiregistry de la PC servidor una referencia remota al ObjetoRemoto. Una vez que la consigue (en realidad obtiene un ObjetoRemoto_Stub), puede hacer las llamadas a sus métodos.

Los métodos se ejecutarán en el Servidor, pero Cliente quedará bloqueado en cada llamada hasta que Servidor termine de ejecutar el método.

2.4.8 rmiregistry.

Antes de registrar el objeto remoto, debemos lanzar, desde una ventana de ms-dos o una shell de Linux el programa rmiregistry. Es importante al arrancarlo que la variable CLASSPATH no tenga ningún valor que permita encontrar nuestros objetos remotos, por lo que se aconseja borrarla antes de lanzar rmiregistry. se lanzaría de la siguiente manera:

En Windows:

```
c:\> set CLASSPATH=
c:\> rmiregistry
```

En Unix o Linux:

```
$ unset CLASSPATH
$ rmiregistry
```

Una vez arrancado rmiregistry, podemos registrar en él nuestros objetos remotos.

2.4.9 Clase Servidor.

Se debe ejecutar un objeto java que instancie y registre el objeto remoto. Este programa java debe indicar cual es el path en el que rmiregistry puede encontrar la clase correspondiente al objeto remoto. Dicho path se da en formato URL, por lo que no admite espacios ni caracteres extraños. Consiste en fijar la propiedad de nombre java.rmi.codebase con el path donde se encuentran los ficheros .class remotos. También se puede indicar la URL "http://un_host/un_path/" si la clase ObjetoRemoto_Stub.class es accesible desde un servidor Web.

Cualquiera que sea el caso, el path que se indique debe ser accesible por el rmiregistry cuando se ejecute. El código para indicar esto es el siguiente:

```
System.setProperty("java.rmi.server.codebase",file:/D:/users/javie
r/prueba_servidor/");
```

Además debe Instanciar una clase remota y luego registrarla en rmiregistry. Para registrarla hay que llamar al método estático rebind() de la clase java.rmi.Naming. Se le pasan dos parámetros. Un nombre para poder identificar el objeto el cual se pasa en formato URL y en el cual se indica el host donde esta el rmiregistry (//host/Object),y una instancia del objeto a registrar. El método rebind() registra el objeto en rmiregistry, de estar ya registrado lo sustituye. Como se muestra en el siguiente código:

```
ObjetoRemoto objetoRemoto = new ObjetoRemoto();
Naming.rebind ("//localhost/ObjetoRemoto", objetoRemoto);
```

2.4.10 Clase Cliente.

El programa que utilice este objeto de forma remota, debe pedir el objeto remoto al servidor de RMI, mediante el método estático lookup() de la clase java.rmi.Naming. A este método se le pasa la URL del objeto. Esa URL es el nombre o IP del host donde se está ejecutando el rmiregistry y por último el nombre con el que se registró, Este método devuelve un Remote, así que debe realizar un Cast a la InterfaceRemota para poder utilizarlo. El objeto que se recibe es realmente un ObjetoRemoto_Stub. Por ello, ObjetoRemoto_Stub.class debe estar en el CLASSPATH del cliente. El código es el siguiente:

```
InterfaceRemota objetoRemoto = (InterfaceRemota)Naming.lookup ("//
host_servidor/ObjetoRemoto");
```

Una vez obtenida la interfaz remota, se puede llamar al método de suma() como si se tratara de una llamada aun método local, como se muestra en el siguiente código:

```
System.out.print ("2 + 3 = ");
System.out.println (objetoRemoto.suma(2, 3));
```

Este ejemplo demuestra el funcionamiento básico de RMI, existen formas de operación mas avanzada como los son el paso objetos Serializables y Remote como parámetros y la carga dinámica de clases, para mayor referencia visite las siguientes URL:

Pasar objetos Serializables y Remote como parámetros en rmi

http://www.chuidiang.com/java/rmi/rmi_parametros.php

Carga dinámica de clases. Seguridad en rmi con RMISecurityManager y java.policy.

http://www.chuidiang.com/java/rmi/rmi_sin_security.php

2.5 JADE (Java Agent Development Framework).

Distribuido por TELECOM Italia bajo la licencia LGPL (Lesser General Public License). Es un Middleware basado en los estándares FIPA (Foundation for Intelligent Physical Agents) [30] de diseño de sistemas multiagente, para el desarrollo de aplicaciones peer to peer.

Los estándares de la FIPA definen a los agentes como entidades de software que están localizadas dentro de una único entorno de trabajo denominado plataforma, donde los agentes se ejecutan, y a través de la cual disponen de un servicio de páginas blancas para buscar a otros agentes, un servicio de páginas amarillas para buscar servicios que otros agentes ofrecen, un módulo de gestión mediante el cual se accede a estas facilidades y por último un sistema de envío y recepción de mensajes mediante el cual el AMS(Agent Management System) puede hacer llegar a su destino los mensajes generados por los

agentes ubicados dentro de la plataforma.

Aparte de los detalles propios de FIPA, JADE [26] incluye la noción de contenedor. Un contenedor es una instancia del entorno de ejecución de JADE. En un contenedor es posible albergar un número indeterminado de agentes. Existe un tipo especial de contenedor denominado principal (main container), del cual debe existir uno y solo uno de estos contenedores por cada plataforma FIPA de agentes y el resto de contenedores de la plataforma, deben subscribirse al principal, por lo que el responsable de ejecutarlos también es responsable de indicar en donde se localiza el contenedor principal [19-21].

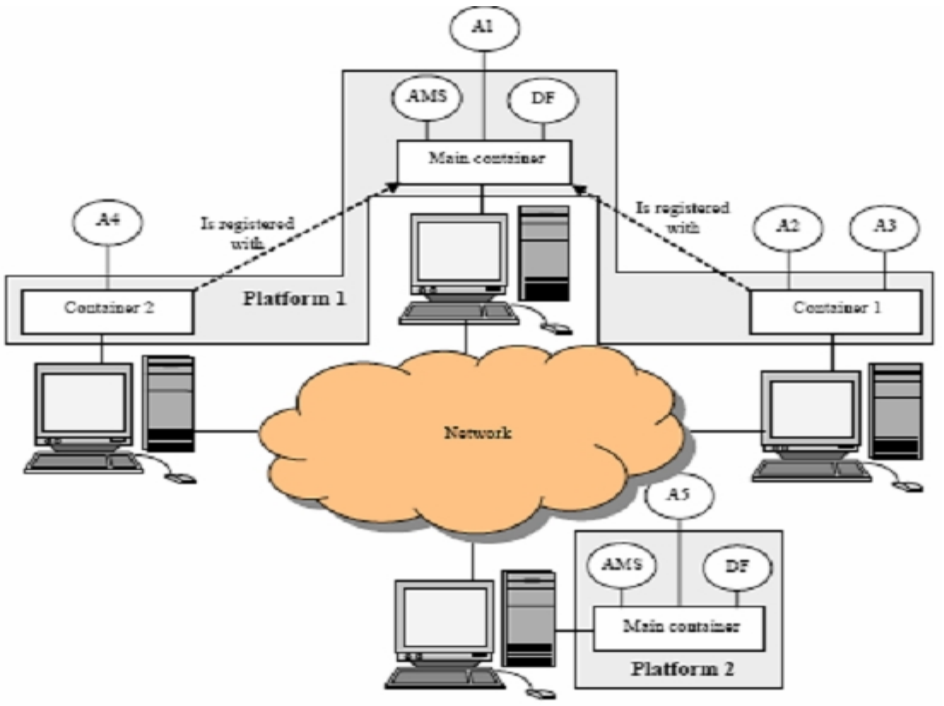

Fig. 2.12 Arquitectura JADE.

2.5.1 Plataforma.

Esta conformada por un conjunto de contenedores, dentro de los cuales existe un denominado principal, el cual tiene la función de registrar los diversos contenedores que forman la plataforma, solo debe haber un contenedor principal dentro de la plataforma,

pero el mecanismo de JADE permite tener varios contenedores secundarios configurados como respaldo del principal, en caso de falla este esquema evita que la plataforma colapse en caso de que el principal tenga algún problema [19-21].

2.5.2 AMS (Agent Management System).

Provee el servicio de nombre llevando el registro de los agentes existentes dentro de la plataforma, administra los agentes dentro de los contenedores que conforma la plataforma. El AMS permite las siguientes acciones [19-21]:

- Iniciar, suspender, reiniciar agentes.
- Matar agentes o contenedores.
- Mandar mensajes.
- Clonar agentes.
- Añadir o quitar plataformas remotas.

2.5.3 DF (Directory Facilitator).

Provee un servicio de páginas amarrillas, esto quiere decir que cada agente dentro de una plataforma en el DF registra los servicios que ofrece, el DF permite realizar las siguientes acciones [19-21]:

- Ver descripciones de los agentes registrados.
- Registrar y desregistrar agentes.
- Modificar registros.
- Buscar descripciones.

2.5.4 Definición de un agente .

Un agente JADE cumple las siguientes características:

- Tiene un nombre único en el entorno de ejecución.
- Se implementa como un único hilo de ejecución (single-thread).

- Tiene un método de inicio (setup) y otro de fin (takeDown).
- El método protegido setup() sirve para inicializar el agente incluyendo instrucciones que especificarán la ontología a utilizar y los comportamientos asociados al agente. Se invoca al comenzar la ejecución del agente.
- El método protegido takeDown() sirve para liberar recursos antes de la eliminación del agente, el cual es invocado cuando se realiza una llamada al método doDelete(), mediante el cual finalizan la ejecución del agente.

2.5.5 Estados de un Agente.

Un agente puede estar en los siguientes estados:

- **Iniciado**: El objeto Agente está creado pero todavía no se ha registrado en el AMS, no tiene nombre ni dirección y tampoco se puede comunicar con otros agentes.
- **Activo**: El Agente está registrado en el AMS, tiene un nombre, una dirección y puede acceder a todas las opciones de JADE.
- **Suspendido**: El Agente está parado. Su hilo de ejecución está detenido y no ejecuta ningún Comportamiento.
- **En espera**: El Agente está bloqueado esperando por algo. Su hilo de ejecución está dormido en un monitor de java y se despertará cuando se cumpla una cierta condición (cuando reciba un mensaje).
- **Desconocido**: El Agente ha sido eliminado. El hilo de ejecución ha terminado y se ha eliminado del registro del AMS.
- **Tránsito**: Un Agente móvil entra en este estado mientras está migrando a una nueva localización. El sistema sigue guardando los mensajes en el buffer hasta que el agente vuelve a estar activo.

2.5.6 Comportamientos de un agente .

Los agentes deben poder ejecutar diferentes tareas al mismo tiempo, JADE tiene un

modelo de agente single threaded y un nivel de scheduling a nivel de comportamientos el cual tiene las siguientes características.

- Cada agente tiene una cola de comportamientos activos.
- El cuerpo de acciones de un comportamiento se programa redefiniendo el método action().
- Cuando el método anterior finaliza, y dependiendo del tipo de comportamiento, el scheduler lo saca de la cola o lo vuelve a colocar al final.
- Un comportamiento puede bloquearse (block()) hasta que lleguen mas mensajes al agente; el bloqueo significa que, cuando action() termina, se le coloca en una cola de bloqueados
- Cuando llega un nuevo mensaje, se le saca de esa cola y se coloca al final de la cola de comportamientos activos

2.5.7 Implementación de un agente.

Para implementar un agente con JADE debemos definir una clase Java que representa al agente la cual hereda de la clase **jade.core.Agent** y que como mínimo implemente el metodo setup() . A continuacion se muestra el ejemplo de un agente comprador.

```
import jade.core.Agent;
public class BookBuyerAgent extends Agent
{
      protected void setup()
      { // Printout a welcome message
            System.out.println("Hallo! Buyer-agent
      "getAID().getName()+          "is ready.'');
      }
}
```

2.5.8 Identificadores de agentes .

En las especificaciones de la FIPA, cada agente debe tener un identificador unico global, la cual contiene información para el envió de mensajes.; la clase jade.core.AID implementa este identificador. La clase Agent incorpora el método getAID() que permite recuperar el identificador del agente. El identificador tiene as siguiente estructura:

```
<nickname>@nombre-plataforma>
```

para obtener el identificador privado de un agente se usa el método getAID() y para obtener el identificador global del agente se utiliza el método getAMS() ambos métodos de la clase agente. Como se muestra en el siguiente fragmento de codigo:

```
System.out.println("Buyer-agent:"+getAID().getName()+" is
ready.")
```

2.5.9 Ejecución de un agente.

Una vez hemos creado el agente, para compilarlo ejecutamos:

```
javac -classpath <jars de jade> BookBuyerAgent.java
```

Ahora ya podemos ejecutarlo. Para eso, tenemos que lanzar la plataforma pasándole como argumentos el identificador del agente y la clase Java en donde se implementa tal como se muestra:

```
java  -classpath <jars de jade>  jade.Boot
comprador:examples.bookTrading.BookBuyerAgent\(linux\)
```

El resultado será el siguiente:

```
May 18, 2009 11:01:23 PM jade.core.Runtime beginContainer
INFO: ----------------------------------
    This is JADE snapshot - revision $WCREV$ of $WCDATE$
    downloaded in Open Source, under LGPL restrictions,
    at http://jade.tilab.com/
----------------------------------------
May 18, 2009 11:01:25 PM jade.core.BaseService init
INFO: Service jade.core.management.AgentManagement initialized
May 18, 2009 11:01:25 PM jade.core.BaseService init
INFO: Service jade.core.messaging.Messaging initialized
May 18, 2009 11:01:25 PM jade.core.BaseService init
INFO: Service jade.core.mobility.AgentMobility initialized
May 18, 2009 11:01:25 PM jade.core.BaseService init
INFO: Service jade.core.event.Notification initialized
May 18, 2009 11:01:25 PM jade.core.messaging.MessagingService
clearCachedSlice
INFO: Clearing cache
May 18, 2009 11:01:25 PM jade.mtp.http.HTTPServer <init>
INFO: HTTP-MTP Using XML parser
com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParse
r
May 18, 2009 11:01:25 PM jade.core.messaging.MessagingService boot
INFO: MTP addresses:
http://ruy:7778/acc
May 18, 2009 11:01:25 PM jade.core.AgentContainerImpl joinPlatform
INFO: ----------------------------------
Agent container Main-Container@ruy is ready.
------------------------------------------
Hallo! Buyer-agent comprador@ruy:1099/JADE is ready.
Target book is linux
```

2.5.10 Especificando tareas mediante comportamientos.

Las tareas o servicios que un agente realiza se especifican en sus comportamientos. El codigo JAVA que implementa esa funcionalidad se incluye en una nueva clase creada para tal efecto, que ha de heredar de la clase jade.core.Behaviours.Behaviour la cual se agrega al agente mediante su método addBehaivour.

Toda clase que herede de la de comportamiento deberá implementar el método action() en donde se debe incluir el codigo de las acciones correspondientes. También el método done() el cual devuelve un boleano y sirve al agente para determinar si el comportamiento ha finalizado o no. Deberá devolver true en ese caso.

Podemos pensar que los behaviours son como los hilos de ejecución JAVA. Igual que las threads en Java, en un agente pueden ejecutarse a la vez tantos comportamientos como sea necesario. Sin embargo, a diferencia de las threads en JAVA, el decidir que comportamiento se ejecuta en cada momento es tarea nuestra. Esto es para que cada agente equivalga únicamente a un unico thread.

Ejemplos de la clase Behaviour:

```
public class OverbearingBehaviour extends Behaviour
{
      public void action()
      {
            while (true)      // do something
      }

      public boolean done()
      {
            return true;
      }
}
```

Toda clase que herede de la de comportamiento deberá implementar el método action() en donde se debe incluir el codigo de las acciones correspondientes. También el método done() el cual devuelve un boleano y sirve al agente para determinar si el comportamiento ha finalizado o no. Deberá devolver true en ese caso.

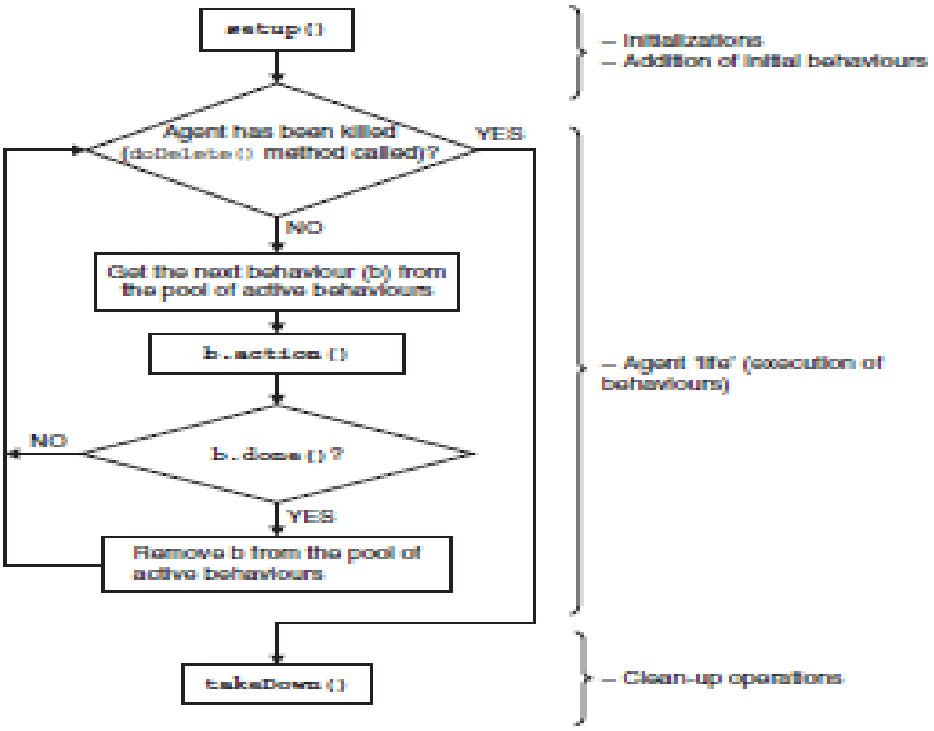

Fig. 2.13 El ciclo de ejecución de un agente.

2.5.11 Tipos de comportamientos.

Podemos agrupar los behaviours JADE en tres grandes grupos:

Comportamientos one-shot.

Tipos de comportamiento los cuales se ejecutan de manera casi instantánea, y solamente una vez; para implementar este comportamiento extendemos de la clase jade.core.behaviours.OneShotBehaviour.

```
public class MyOneShotBehaviour extends OneShotBehaviour
{
      public void action()
      {
            // perform operation X
      }
}
```

Comportamientos cíclicos.

Son aquellos que nunca son sacados del conjunto de comportamientos del agente y cuyo método action() siempre ejecuta el mismo codigo. Por lo tanto, nunca finalizan. para implementar este comportamiento extendemos de la clase jade.core.behaviours.CyclicBehaviour.

```
public class MyCyclicBehaviour extends CyclicBehaviour
{
      public void action()
      {
            // perform operation Y
      }
}
```

Comportamientos genéricos.

El codigo que se ejecuta en ellos depende del estatus del agente, y eventualmente finalizan su ejecución.

```
public class ThreeStepBehaviour extends Behaviour
{
      public void action()
      {
            // perform operation X
      }
```

```
}
```

Otro tipo de comportamientos JADE son los compuestos, que permiten combinar comportamientos previamente definidos de manera conveniente para aplicarles un orden de ejecución determinado como secuencial con SequentialBehaviour, en paralelo con ParallelBehaviour y guiados mediante un autómata finito determinista con FSMBehaviour.

Otros tipos de comportamientos interesantes son WakerBehaviour y TickerBehaviour. Tienen en común que su ejecución es diferida. Se invocan y guardan hasta que se ha cumplido un determinado time-out.

Ejemplo de agente que utiliza un WakerBehaviour, el cual se ejecuta después de 10 segundos.

```
public class MyAgent extends Agent
{
      protected void setup()
      {
            System.out.println(``Adding waker behaviour'');
            addBehaviour(new WakerBehaviour(this, 10000)
            {
                  protected void handleElapsedTimeout()
                  {
                        // perform operation X
                  }
            });
      }
}
```

2.5.12 Comunicación entre agentes

Una de las mas importantes características de los agentes JADE es la habilidad para comunicarse, el paradigma de comunicación utilizado se basa en un esquema en un esquema asíncrono de paso de mensajes, cada agente tiene un buzón de Mensajes, cuando un mensaje llega el agente es notificado para que sea procesado por el agente.

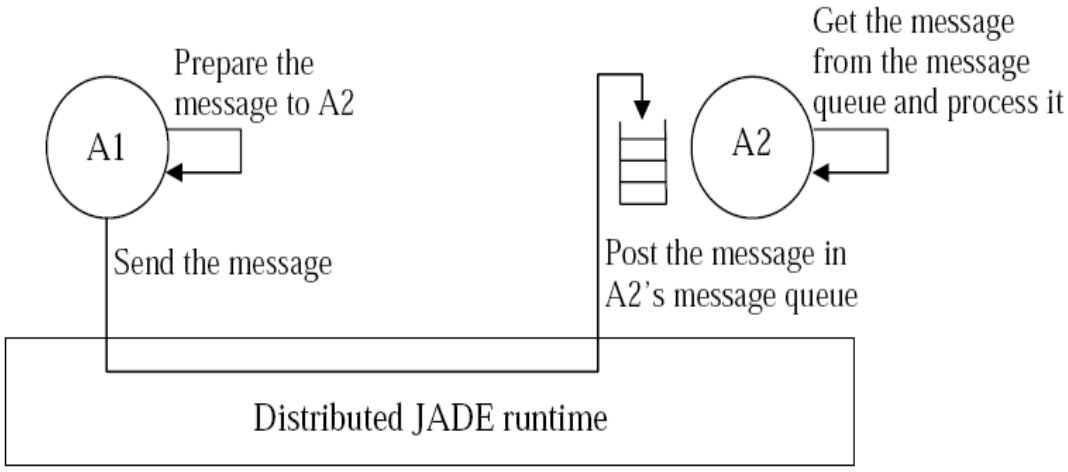

Fig. 2.14 Esquema de transferencia asíncrona de mensajes en JADE

Los mensajes intercambiados por JADE se basa en el formato especificado en el ACL definido por la FIPA, este formato comprende los siguientes campos:

- el remitente
- la lista de destinatario
- la intención del mensaje
- el contenido del mensaje
- el lenguaje del contenido
- la ontología del mensaje

La clase jade.lang.acl.ACLMessage implementa un conjunto de métodos para el manejo

de los campos de los mensajes, el envió de un mensaje entre agentes es simple, solo es necesario rellenar los campos del mensaje e invocar el método send() del agente, como se muestra en el siguiente ejemplo:

```
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);
msg.addReceiver(new AID("Peter", AID.ISLOCALNAME));
msg.setLanguage("English");
msg.setOntology("Weather-forecast-ontology");
msg.setContent("Today it's raining");
send(msg);
```

2.5.13 localizar a los agentes mediante directorio.

La arquitectura abstracta de FIPA define un elemento opcional denominado facilitados de directorio (DF). La plataforma JADE incorpora este elemento en forma de clases Java para poder inscribir nuestros agentes en el sistema y para poder encontrar otros agentes que previamente se han suscrito a ellos y a sus servicios. El siguiente código muestra como un agente se registra en el DF.

```
protected void setup()
{
      ...
      // Register the book-selling service in the yellow pages
      DFAgentDescription dfd = new DFAgentDescription();
      dfd.setName(getAID());
      ServiceDescription sd = new ServiceDescription();
      sd.setType(``book-selling'');
      sd.setName(``JADE-book-trading'');
      dfd.addServices(sd);
      try
      {
            DFService.register(this, dfd);
      }
```

```
        catch (FIPAException fe)
        {
              fe.printStackTrace();
        }
        ...
}
```

El agente crea un objeto de la clase DFAgentDescription que responde a una entrada en el directorio que hace referencia a los datos de un agente. Un objeto de este tipo puede, a su vez, contener uno o más descripciones de servicios, objetos de la clase ServiceDescription, la descripción de un servicio contiene una serie de parámetros como el nombre y el tipo.

Cuando el agente finaliza su ejecución es requerido darlo de baja dentro del DF.

```
protected void takeDown()
{     // Deregister from the yellow pages
      try
      {
            DFService.deregister(this);
      }
      ..............
}
```

Los agentes deben ser capaces de encontrar a los agentes en base de los servicios que ofrecen. Para ello se utiliza el método DFService.search().

```
public class BookBuyerAgent extends Agent
{
      // The title of the book to buy
      private String targetBookTitle;
      // The list of known seller
```

```
agents private AID[] sellerAgents;
// Put agent initializations here
protected void setup()
{
      ........
      addBehaviour(new TickerBehaviour(this, 60000) {
      protected void onTick()
      {
            // Update the list of seller agents
            DFAgentDescription template = new
            DFAgentDescription();
            ServiceDescription sd = new
            ServiceDescription();
            sd.setType(``book-selling'');
            template.addServices(sd);
      try
      {
            DFAgentDescription[] result =

            DFService.search(myAgent,template);
            sellerAgents = new AID[result.length];
            for (int i = 0; i < result.length; ++i)
            {
                  sellerAgents[i] = result.getName();
            }
      ..............
      ........................
```

Capitulo III

Descripción de la Investigación.

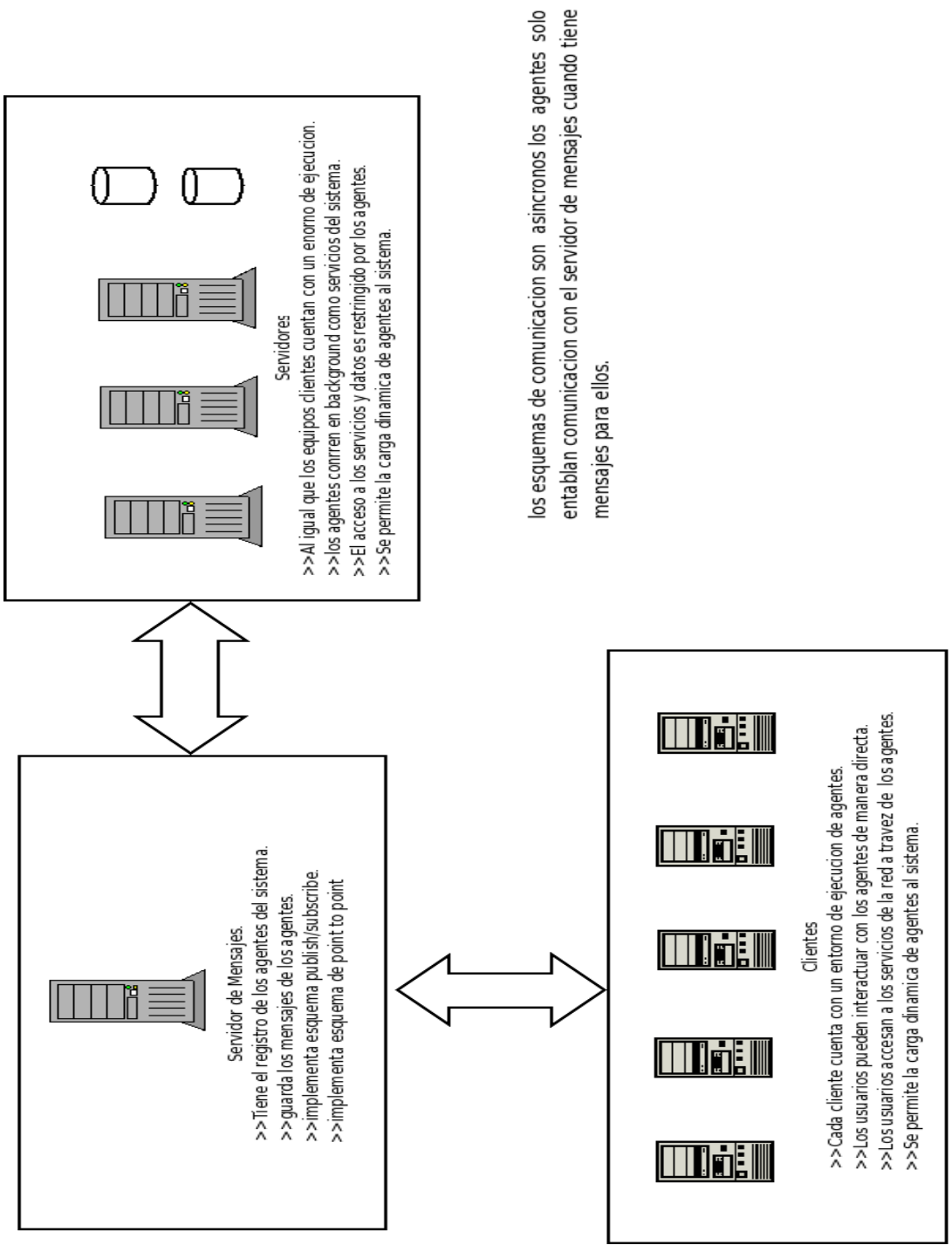

Fig. 3.1 Esquema general.

3.1 Descripción del Framework.

La idea principal sobre la cual se base la construcción del framework, es la de facilitar el desarrollo de aplicaciones distribuidas, mediante el uso de agentes. Como se muestra en la figura 3.1 el framework esta diseñado para ambientes de red con múltiples servidores y equipos clientes, brindando un conjunto de herramientas que implementa la comunicación entre agentes, liberando de esta tarea al programador permitiéndole enfocarse en el modelado del sistema.

Objetivos.

- Facilitar el desarrollo de sistemas multiagentes.
- Implementar esquema de comunicación asíncrono.
- Administrar los agentes registrado en el sistema.
- Definición de grupos para envió de mensajes (publish-subscribe).
- Publicación dinámica de agentes en el sistema.
- Implementa un ambiente homogéneo de ejecución para agentes.

3.2 Casos de uso del Framework.

Las funciones básicas que debe cubrir el framework se muestran en el siguiente diagrama de casos de uso.

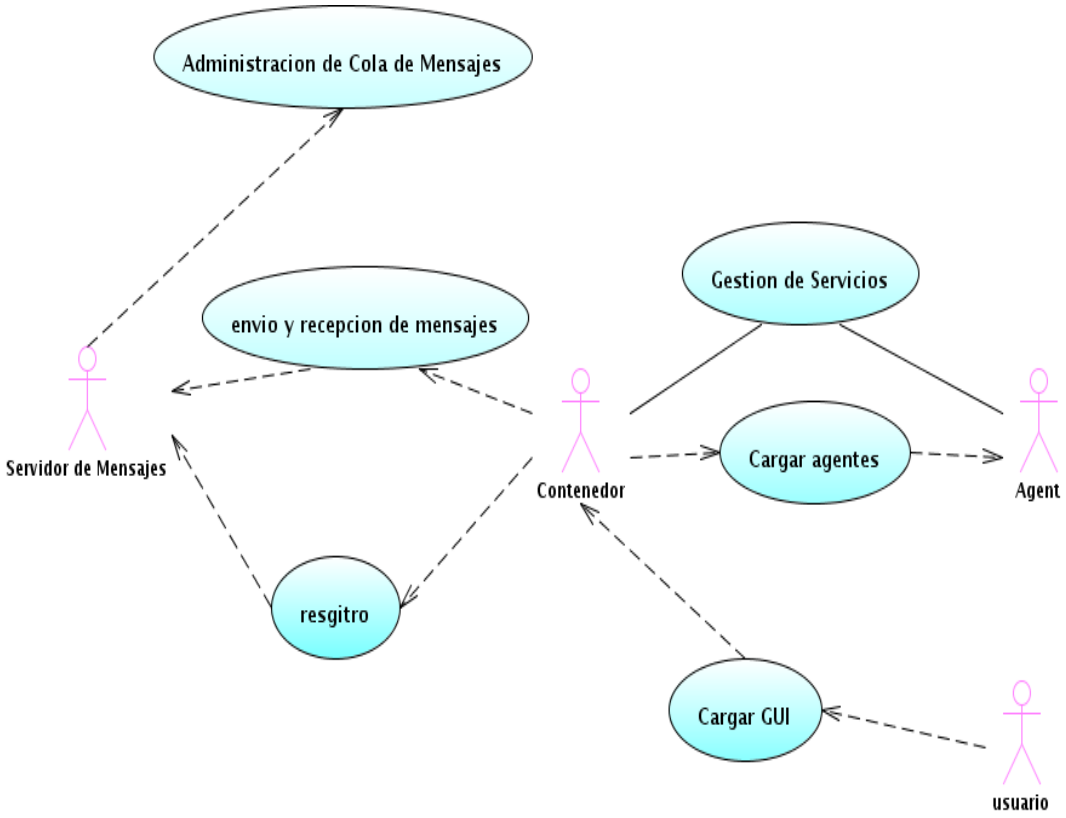

Fig. 3.2 Diagrama de Caso de uso.

3.2.1 Registro.

En el servidor de mensajes se registran los agentes y los contenedores de la red. La información de registro es utilizada para la administración de las colas de mensajes, así como para la notificación de eventos remotos enviada a los contenedores.

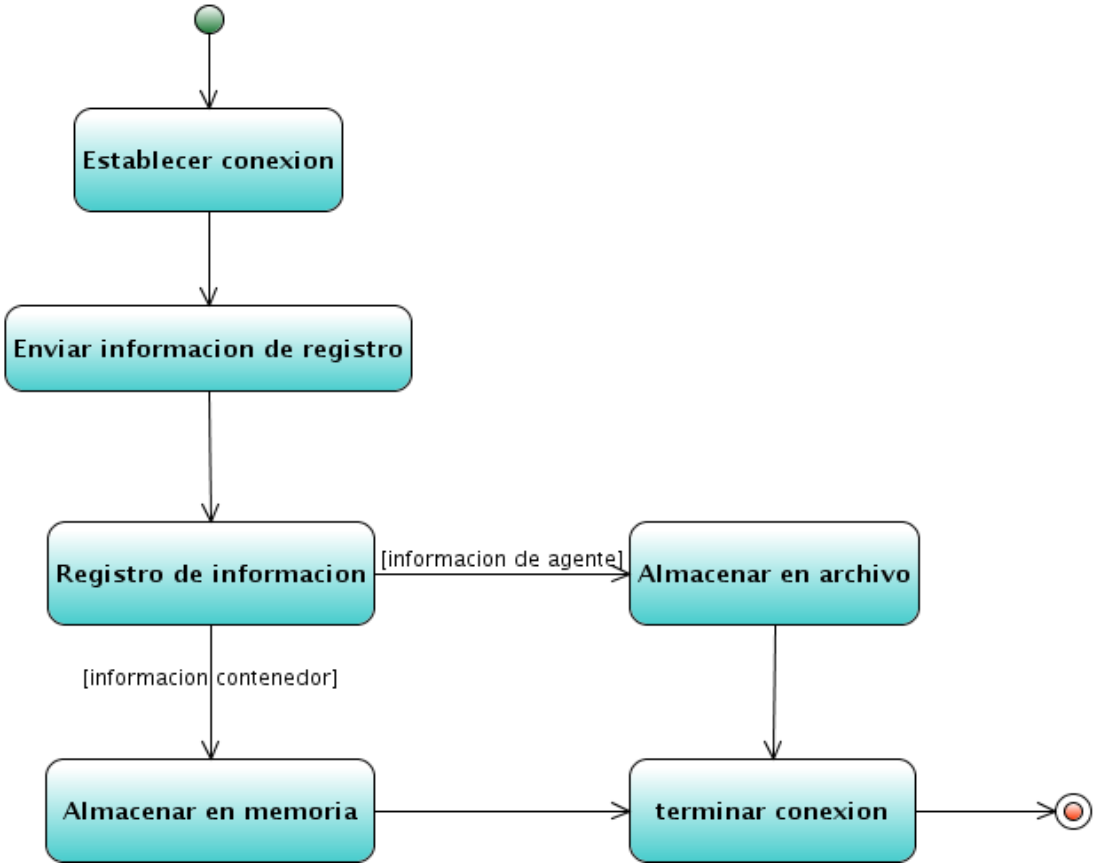

Fig. 3.3 Diagrama de actividades de registro.

Como se muestra en el diagrama de actividades de la figura 3.3 el registro se lleva acabo a través de la red, en este caso el que entabla la conexión es el contendor enviando su información que lo identifica dentro de la red y los identificadores de los agentes que hospeda, dependiendo de la información de registro se almacena en memoria en el caso de los contenedores o en archivo xml paro los agentes.

3.2.2 Administración de cola de mensajes.

Esta actividad es el núcleo del sistema ya que coordina la entrega de mensajes a los agentes registrados.

El flujo de la administración de la cola de mensajes se muestra en el diagrama de actividades de la figura 3.4, existen 2 eventos a través del cual se acceden a las colas de mensajes, durante la recepción de un nuevo mensaje o cuando se recibe la solicitud de descarga de mensajes.

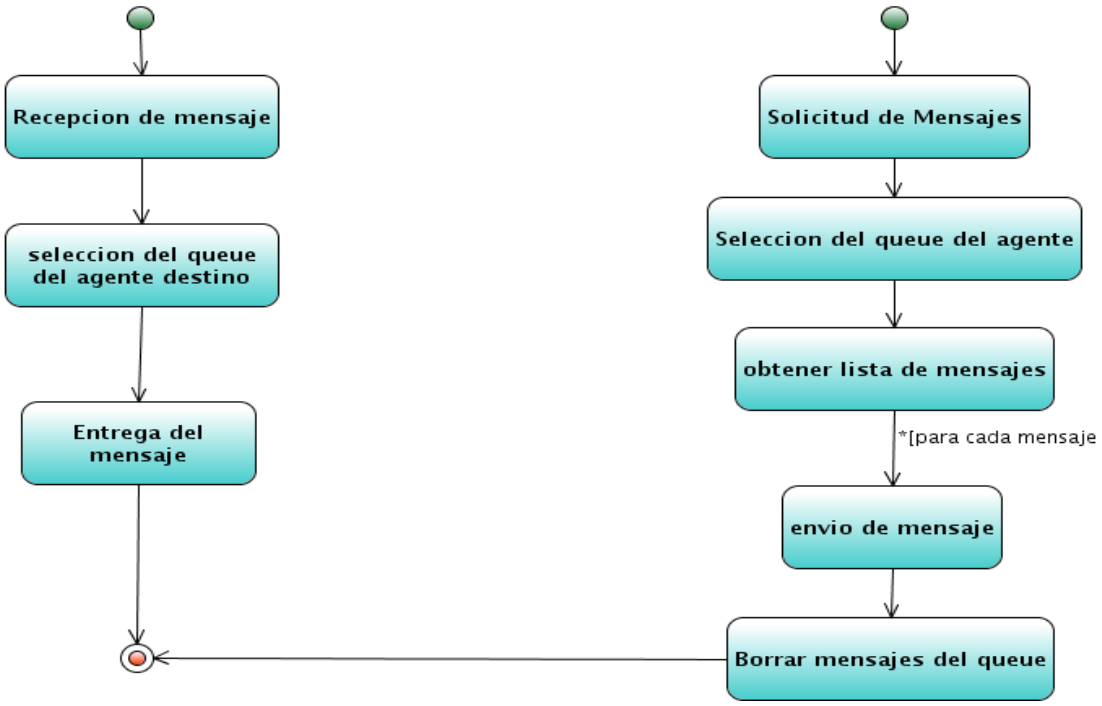

Fig. 3.4 Diagrama de actividades Administración de cola de mensajes

3.2.3 Envió y recepción de mensajes.

Para minimizar la carga al servidor de mensajes, los agentes no se conectan directamente para el envió y recepción de mensajes, estas tareas las realiza el contendor el cual coordina internamente la realización de estas tareas.

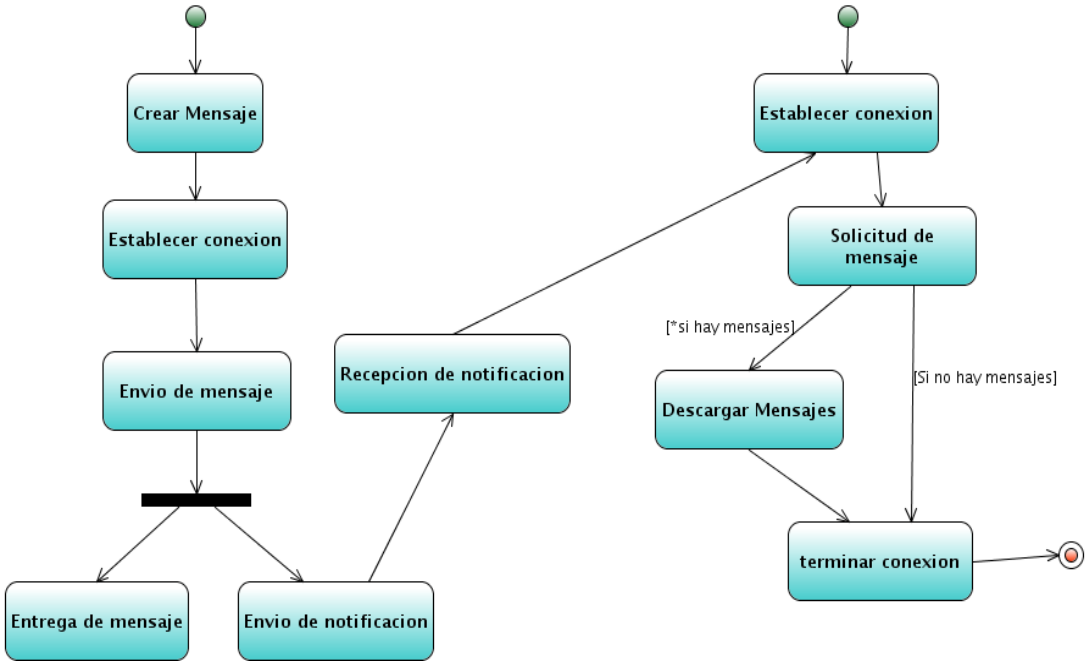

Fig. 3.5 Diagrama de actividades Envió y recepción de mensajes

Como se muestra en la figura 3.5 el flujo de envió de mensajes inicia con la creación del mensajes el cual es generado por el agente, pero no es enviado directamente por el, la responsabilidad del envió es del contenedor, el cual entabla la conexión con el servidor realizando la entrega de los mensajes, disparando el evento de recepción, los contendores recibirán una notificación y se conectaran al servidor para realizar la descarga de los mensajes, la conexión de descargar por parte de los contendores se debe realizar cada determinado tiempo independientemente de los eventos lanzados.

3.2.4 Carga de agentes.

Mediante esta funcionalidad, se realiza la carga dinámica de agentes al entorno de ejecución del sistema, sin tener que detener el contenedor para agrega un nuevo elemento, como se muestra en el diagrama 3.6. El contenedor revisa la ruta y carga todos los archivos de definición de agentes que encuentre, es responsabilidad del contenedor

cargar e iniciar los agentes, además guarda una referencia de la interfaz gráfica con la cual el usuario podrá comunicarse con el en caso de tener definida una; cada 10 segundos revisa la ruta para verificar si han sido publicados nuevos agentes.

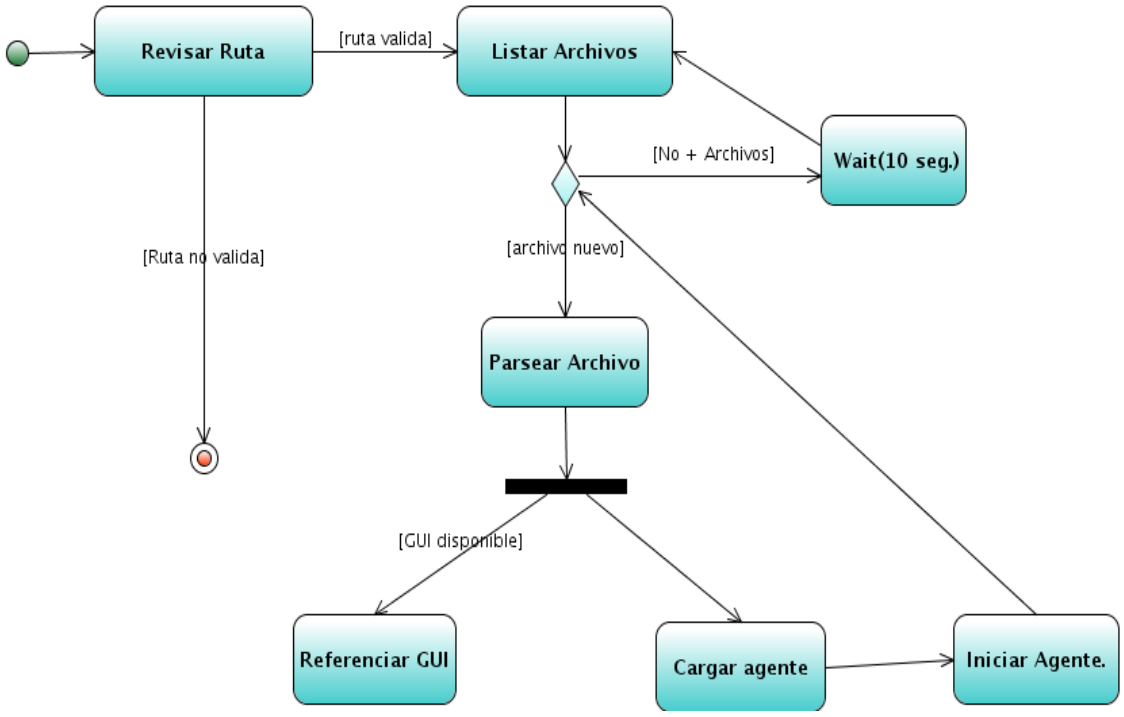

Fig. 3.6 Diagrama de actividades carga de agentes.

3.2.5 Gestión de servicios.

Esta actividad es controlada por el contenedor, ya que este brinda una serie de interfaces que los agentes necesitan para el envío y recepción de mensajes, el flujo de esta actividad se muestra en la figura 3.7

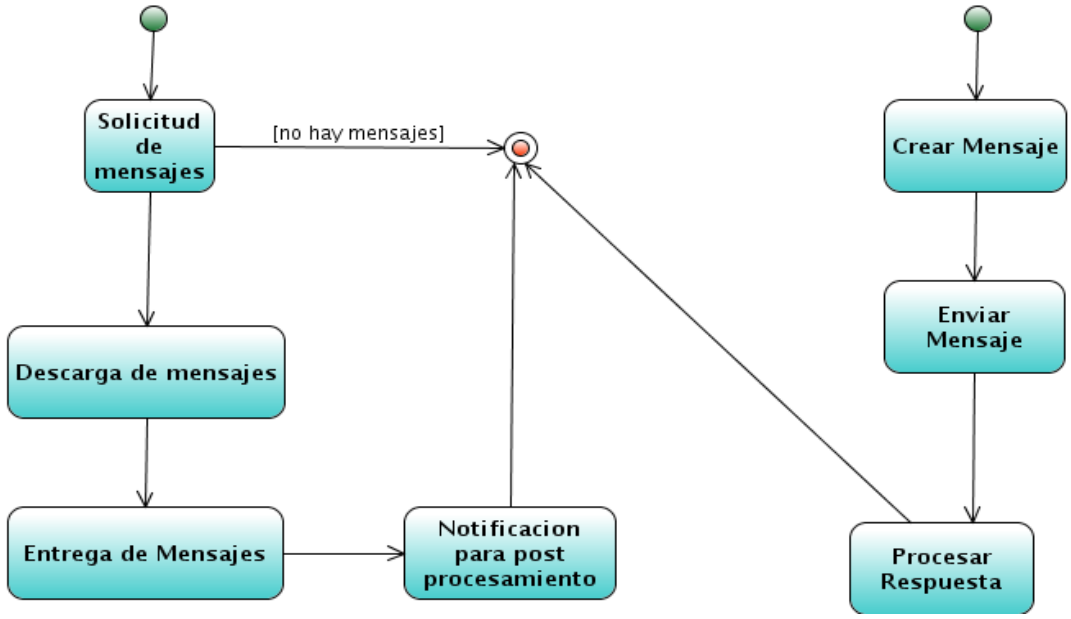

Fig. 3.7 Diagrama de actividades gestión de servicios.

3.2.6 Cargar GUI.

Mediante esta funcionalidad es como el usuario podrá comunicarse con los agentes del contenedor permitiendo así diseñar aplicaciones en donde el usuario pueda usar distintas aplicaciones gráficas que sean atendidas por los agentes del sistema. El contenedor guarda una lista de las interfaces gráficas disponibles, poniéndolas a disposición del usuario que puede cargar las aplicaciones que son atendidas por los agentes.

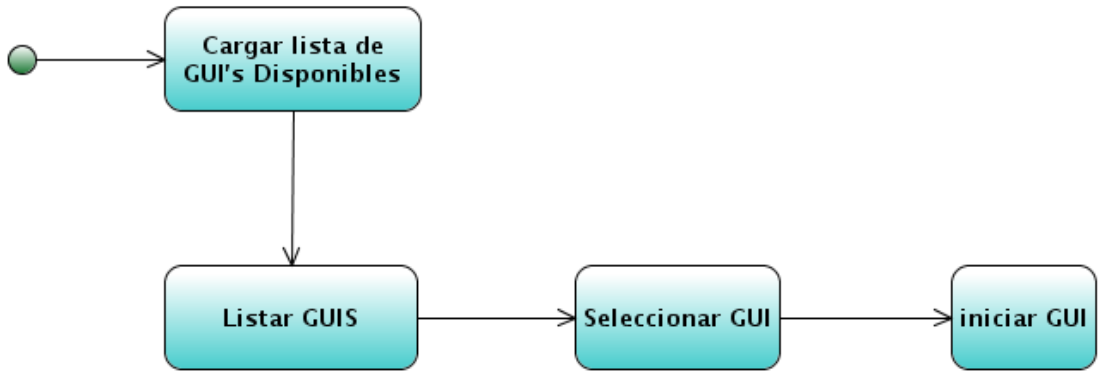

Fig. 3.8 Diagrama de actividades cargar Intefaz gráfica

3.3 Implementación del sistema.

Como se muestra en el diagrama 3.9 el framework esta compuesto por varios objetos los cuales tienen responsabilidades bien definidas, facilitando futuras mejoras y modificaciones al mismo.

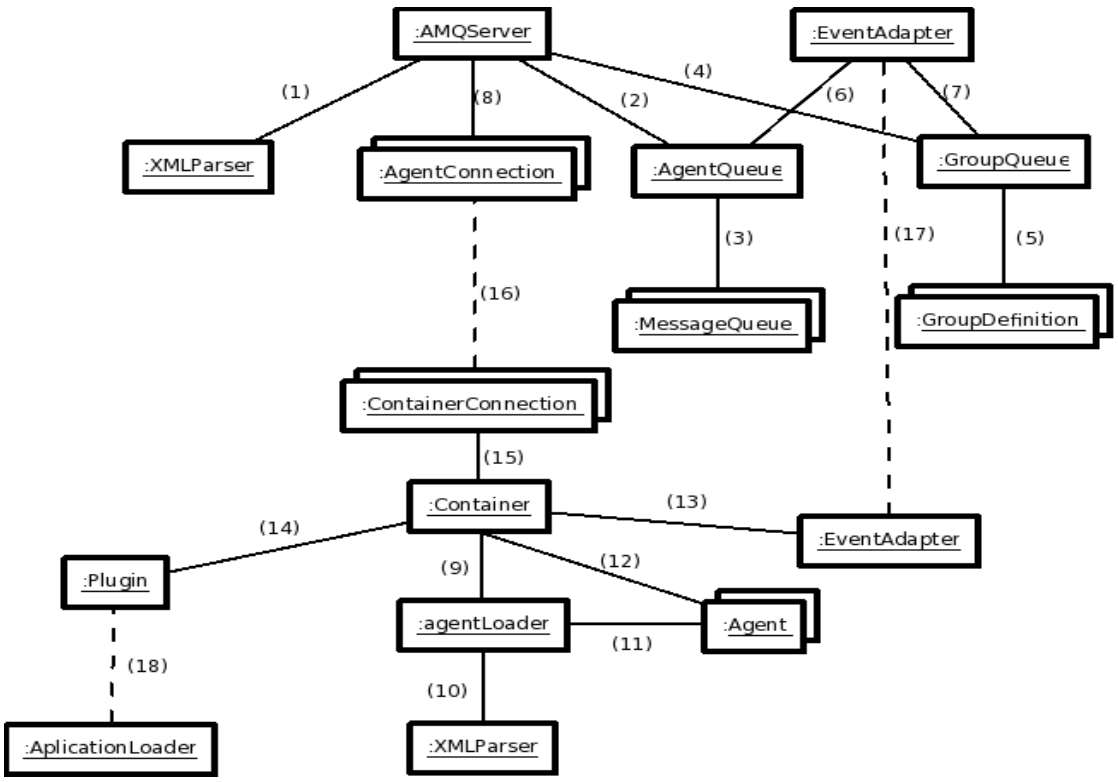

Fig. 3.9 Diagrama general del funcionamiento del sistema.

3.3.1 Descripción de la Interacción entre los componentes.

Utilizando el diagrama de la figura 3.9 se procede a explicar la interacción entre los componentes que conforman el framework.

1.-El AMQServer instancia al XMLParser para cargar los archivos de configuración donde se encuentran definidos los agentes y los grupos para la administración de colas de mensajes.

2.-El AMQServer instancia la estructura que guarda la referencia entre los agentes y su cola de mensajes, esta estructura se llama AgentQueue.

3.-El AgentQueue guarda una referencia a una cola de mensajes (MessageQueue) por cada agente dado de alta en el sistema, la cual internamente implementa las operaciones para su administración.

4.- El AMQServer instancia la estructura que guarda referencia de los grupos para envió

de mensajes, por cada grupo definido en el sistema guarda una referencia a un GroupDefinition.

5.-El GroupDefinition mantiene una lista de los ID de los agentes pertenecientes a un grupo, así como la operaciones para administrar el grupo.

6.-Cada vez que se recibe un mensaje para un agente se dispara el evento, el EventAdapter se encarga de manejarlo.

7.-Cada vez que se recibe un mensaje para un Grupo se dispara el evento, el EventAdapter se encarga de manejarlo.

8.-Cuando el AMQServer recibe una conexión de un contenedor crea un AgentConnection para manejarlo.

9.- El Container crea el AgentLoader que es el responsable de cargar los agentes en memoria.

10.-El AgentLoader se ayuda del XMLoader para la lectura de los archivos de definición de agentes para cargarlos en memoria.

11.-El container carga y ejecuta los agentes en memoria.

12. -Los agentes se comunican directamente con el contenedor para el envió y recepción de mensajes.

13.- El Container crea un EventAdapter para manejar las notificaciones remotas de entrega de mensajes, cuando un mensaje es entregado en el servidor este objeto recibe la notificación y avisa al contenedor para realizar la descarga de l mensaje.

14.-El Container crea el objeto Plugin que es el servicio que permite que conecten para cargar las interfaces graficas de los agentes del sistema.

15.-El Container crea el Objeto ContainerConnection que mantiene la información para establecer la comunicación con el AMQServer.

16.-La comunicación entre el AgentConnection y el ContainerConnection se entabla de manera asíncrona, esto quiere decir que el canal de comunicación se enlaza cuando va a realizar alguna tarea con el AMQServer, este esquema se diseño para aminorar la carga en el servidor de Mensajes, al solo tener conexiones activas de los contenedores que están

enviando o descargando mensajes.

17.-El EventAdapter del AMQServer envía notificaciones cuando se recibe un mensaje para un agente registrado del sistema, los EventAdapter de los contenedores reciben y procesan estas notificaciones; este esquema es el que permite que los contenedores solo se conecten para descargar sus mensajes.

18.-Mediante el AplicationLoader los usuarios pueden interactuar con el contendor para determinar las interfaces graficas disponibles en el Container y así cargar las aplicaciones para interactuar con los agentes.

3.3.2 División en paquetes del framework.

El sistema fue dividido en 3 paquetes para dividir las funciones, el AMQServer es el paquete donde esta definida las funciones de administración de agentes del sistema, así como el manejo de las colas de mensajes. El AgentContainer contiene el ambiente de ejecución y desarrollo para agentes; y el AMQCommon contiene las clases que son de uso común entre ambos paquetes.

3.4 Descripción del paquete AMQServer

Debido a que se necesita un esquema de comunicación asíncrona, en el cual los agentes puedan enviar información a los demás agentes aunque estos actualmente no estén en línea, se maneja el concepto de un Middleware Orientado a Mensajes, el cual nos permite tener un esquema de comunicación de mensajería asíncrona uno a uno, como uno a muchos. Este esquema se puede visualizar como un una oficina postal con múltiples buzones donde se concentran los mensajes, los agente envían diversos mensajes y el sistema se encarga de entregarlos en los buzones correspondientes, además de notificar a sus destinarlos que ha recibido mensaje; en caso de que los agentes no estén en línea los mensajes seguirán siendo almacenados esperando a que sean descargados, en la figura 3.10 se muestra el diagrama de clases del paquete AMQServer.

Para ver el diagrama consulte los anexos.

Fig. 3.10 Diagrama de Clase AMQServer.

Para llevar un control de los agentes, grupos y contenedores que forman parte del sistema, se manejan diversas estructuras que almacenan la información necesaria para el manejo de los mensajes; la estructura que guarda información de los agentes relaciona los identificadores con los cuales fueron registrados con su cola de mensajes.

Para el manejo de grupos se mantiene una relación de los identificadores de los agentes pertenecientes a cierto grupo, brindado así la facilidad de que el esquema de envió de mensajes sea el mismo para un esquema uno a uno, como a un grupo, ya que internamente el sistema se encarga de la redistribución entre los miembros del grupo.

Debido a que el sistema esta diseñado para que los agentes se encuentren localizados dentro de un entorno de ejecución, denominado contenedor, el servidor almacena la información de los contenedores para realizar las notificaciones de recepción de nuevos mensajes.

3.4.1 Descripción de componentes de AMQServer.

A continuación se realiza una descripción detallada de los componentes que conforman el servidor de Mensajes.

Server.AMQServer

Es la clase principal del servidor de mensajes, tiene la función principal de cargar los componentes del sistema, así como de coordinar la comunicación entres los mismos, además tiene las estructuras necesaria para la administración de los mensajes.

Atributos.

private Vector agents.

Guarda la referencia a las conexiones realizadas por los contenedores que forman parte de la red.

private AgentQueue queue.

Estructura donde se almacena los mensajes de los agentes del sistema.

private RegistrationMap registrationMap.

Estructura donde se almacena la información de los contenedores dados de alta en el sistema.

private EventAdapter eventAdpater.

Componente que se encarga de manejar los eventos del sistema.

private XMLParser parser.

Componente que se encarga de cargar las configuraciones del sistema que se encuentra definidas dentro de archivos xml (definición de agentes y grupos).

private GroupQueue groups.

Estructura donde se almacena la información de los elementos pertenecientes a los grupos registrados en el sistema.

Métodos.

public AMQServer().

Constructor de la clase.

public void start().

Registra el socket en escucha del sistema y se queda en espera de las conexiones de los clientes.

synchronized public void removeClient(AgentConnection client).

Este método elimina la referencia al hilo de ejecución encargado del manejo de la

conexión de un contenedor del sistema.

synchronized public Object getMsgQueue(Object agentID).

Permite obtener la referencia a la cola de mensajes de un agente en específico.

synchronized public Object getGroup(Object group).

Obtiene la referencia a la estructura donde se almacena la información de los grupos registrados en el sistema.

synchronized public Object getGroupsFrom(Object agentID).

Regresa una lista de los grupos a los que pertenece un agente.

synchronized public boolean registration(RegistrationInformation ri).

Registra a los contenedores en el RegistrationMap.

synchronized public void notifyMsg(QueueEvent qe).

Este método se invoca cuando un mensaje nuevo ha llegado, el QueueEvent contiene la información necesaria para determinar a que agente le pertenece el mensaje.

synchronized public boolean addAgent2XML(AgentDefinition ad).

Este método permite registrar agentes de manera permanente al archivo de configuración de agentes del sistema.

public static void main(String[] args).

Método con el que inicia la ejecución del AMQServer.

Interfaces.

Server.AMQSInterface

Server.AgentConnection

Este componente se crea cada vez que se conecta un contenedor, su función principal es la de procesar los mensajes y realizar las tarea correspondiente a cada performative, las operaciones que realiza son las sig. Dar de alta contenedores, registrar nuevos agentes, recepción y envió de mensajes.

Atributos.

private Socket coneccion.

Es el socket de conexión con el contenedor.

private ObjectInputStream entrada.

Es la referencia al Stream de entrada del enlace de comunicación.

private ObjectOutputStream salida.

Es la referencia al Stream de salida del enlace de comunicación.

private AMQSInterface server.

Mantiene una referencia al AMQSever a través de la interfaz AMQSInterface para poder acceder a los métodos definidos.

private Wrapper response.

Utiliza este componente como recipiente para la recepción y envió de mensajes.

private MessageQueue queue.

Utiliza este objeto para referenciar la cola de mensajes de los agentes.

Métodos.

public AgentConnection(AMQSInterface server).

Constructor de la clase, se le pasa una referencia del AMQServer a través de una interfaz AMQSinterface para poder acceder a las estructuras de control que coordina el servidor, así como para comunicarse con las otras clases del servidor.

public void setConnection(Socket clientSocket).

Este método se utiliza para configurar el enlace de comunicación.

public void run().

Este método recibe mensajes a través del socket, redirecciona el mensaje para su procesamiento; implementa este método de la interfaz Runnable para poder ejecutarse en un hilo independiente.

private void procesar (Wrapper msg).

Dependiendo del performative del mensaje invoca el método que lo procesara.

public void requestMsg(Wrapper msg).

Cuando el performative del mensaje es REQUEST obtiene la referencia a la cola de mensajes del agente que esta solicitando sus mensajes y lo envía al contenedor en el que radica.

public boolean sendMsg(Wrapper msg).

Este método envía los mensajes a través de la conexión establecida.

private void sendResponse(String sender,String receiver,Object msg).

Su función es la de mandar respuestas a los mensajes recibidos por el contenedor.

public void giveMsg(Wrapper msg).

Cuando el performative del mensaje es TELL obtiene la referencia a la cola de mensajes

del agente y agrega el nuevo mensaje.

private void processCommand(Wrapper msg).

Cuando el performative del mensaje es COMMAND ejecuta el comando enviado por el contenedor, actualmente el único comando que procesa es ADDAGENT el cual se utiliza para registrar un agente al sistema.

private void processResponse(Wrapper msg).

Actualmente es un método vacío, el objetivo es poder procesar repuestas de los mensajes que envié el AMQServer a los contenedores del sistema. Como por ejemplo algún comando a nivel framework.

private void subscribe(Wrapper msg).

Cuando el performative del mensaje es SUBSCRIBE registra la información del contenedor en el sistema.

private void publish(Wrapper msg).

Cuando el performative del mensaje es INFORM este obtiene una lista de los miembros integrantes del grupo al cual va dirigido y les hace entrega del mismo, para optimizar el uso de memoria solo existe una sola instancia y las diversas colas de mensaje hacen referencia a la misma instancia, implementación del esquema publish-subscribe.

public void cerrar().

Este método cierra el enlace de comunicación.

Interfaces.

Java.lang.Runnable

Server.XMLParser

Este componente tiene la función de cargar los archivos de configuración del sistema los cuales se encuentran en xml, actualmente se utilizan 2 archivos SystemAgents.xml y SystemGroups.xml dentro de los cuales se encuentran registrados loa gentes y grupos del sistema, a continuación se describe el contenido de cada uno:

SystemAgents.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Document   : SystemAgents.xml
    Created on : January 12, 2009, 8:34 PM     Author     : ruy
-->
<SystemAgents>
      <Agent>
            <ID>cid</ID>
            <Cert>231182</Cert>
            <LeaseTime></LeaseTime>
      </Agent>
</SystemAgents>
```

Tag	**Descripción**
<SystemAgents>	es el inicio del archivo de configuración,
<Agent>	se utiliza para dar inicio a la definición de un agente dentro del sistema, el tag, el archivo puede contener múltiples definiciones de agentes.
<ID>	Es el identificador único dentro del sistema para un agente.
<Cert>	Actualmente no se utiliza, el objetivo es brindar un medio de identificación del agente.
<LeaseTime>	actualmente los agentes son registrados de manera permanente, el objetivo de este tag es poder registrar a los agentes por periodos de

tiempo.

SystemGroups.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    Document   : SystemGroups.xml
    Created on : March 8, 2009, 1:15 PM
    Author     : ruy
-->
<SystemGroups>
      <Group>
                  <ID>TEST</ID>
                  <Members>
                        <Agent>cid</Agent>
                        <Agent>tania</Agent>
                  </Members>
      </Group>
</SystemGroups>
```

Tag	**Descripción.**
<SystemGroups>	inicio del archivo de configuración.
<Group>	es el inicio de la descripción del grupo, el archivo puede contener múltiples definiciones de grupos.
<ID>	Es el identificador único para el grupo.
<Members>	Es el inicio de la lista de los miembros del grupo.
<Agent>	dentro de la tag <Members> pueden ver una o mas apariciones de esta tag la cual hace referencia al identificador único de un agente definido dentro del archivo SystemAgents.xml.

Atributos.

static public int AGENTS=0.

Esta constante se usa para indicarle al método loadXML que tipo de archivo va a cargar, en este caso seria el archivo de configuración de agentes SystemAgents.xml.

static public int GROUPS=1.

Esta constante se usa para indicarle al método loadXML que tipo de archivo va a cargar, en este caso seria el archivo de configuración de grupos SystemGroups.xml.

private SAXParser sparser.

Este atributo hace referencia a componente org.apache.xerces.parsers.SAXParser el cual se encarga de realizar el procesamiento de lo archivos xml.

private AgentDefinition agent.

Mantiene la referencia a los datos que se cargaron del archivo SystemAgents.xml que definen un agente para el servidor de mensajes.

private GroupDefinition group.

Mantiene la referencia a los datos que se cargaron de archivo SystemGroups.xml que definen un grupo para el servidor de mensajes.

private Vector agents.

Mantiene las referencias a las definiciones de los agentes que han sido cargadas por el parser

private Vector groups.

Mantiene la referencia a las definiciones de los grupos que han sido cargadas por el parser.

private StringBuffer value.

Variable Global para uso interno de los métodos.

private int set=0.

Variable Global para uso interno de los métodos.

private int type=0.

Variable Global para uso interno de los métodos.

Métodos.

public Iterator loadXML(int type).

Carga el archivo xml de configuración correspondiente dependiendo del tipo indicado.

public boolean readXML(String path) .

Realiza la carga del archivo xml.

public void startDocument() .

Método sobre escrito de la clase DefaultHandler, que permite manejar cuando se inicia el archivo xml.

public void endDocument() .

Método sobre escrito de la clase DefaultHandler, que permite manejar cuando se llega al fin del archivo xml.

public void startElement(String uri, String localName, String qName, org.xml.sax.Attributes attributes).

Método sobre escrito de la clase DefaultHandler, que permite manejar cuando se lee un tag de inicio de elemento.

public void endElement(String uri,String localName,String rawName).

Método sobre escrito de la clase DefaultHandler, que permite manejar cuando se lee un tag de finalización de elemento.

public void characters (char ch[], int start, int length).

Método sobre escrito de la clase DefaultHandler, que permite obtener los valores de los elementos.

public boolean addAgent(AgentDefinition ad).

Permite agregar definiciones de agentes al archivo SystemAgents.xml.

public boolean delAgent(AgentDefinition ad).

Aun no esta implementado, el objetivo es poder eliminar definiciones de agentes del archivo SystemAgents.xml.

Herencia.

org.xml.sax.helpers.DefaultHandler

Server.EventAdapter

Este componente se encarga de manejar los eventos cuando un mensaje llega, como cuando el registro de un contenedor se ha efectuado, internamente mantiene un registro de las IP de los contenedores para realizar el envío de las notificaciones de recepción de nuevos mensajes. Para mejorar el funcionamiento de la notificación se necesita agregar un método que se realice cuando un contenedor sea dado de baja para mantener actualizada la lista. el esquema de envio de notificaciones se basa en el modelo de eventos de java [23], el cual fue extendido para el vio a traves de la red, se selecciono el protocolo UDP para mimizar la carga al servidor de mensaje, ya que es mas facil la construccion y manejo de paquetes de este protocolo, ademas que el diseño del framework esta orientado para construccion de aplicacion con esquemas de comunicacion

asincronas.

Atributos.

private Vector queueListener.

Mantiene el listado de direcciones IP de los contenedores registrados en el sistema.

Métodos.

public EventAdapter().

Constructor de la clase.

public byte [] toByteArray(Object evt).

Debido a que la comunicación se realiza mediante socket UPD los objetos deben ser convertidos a un arreglo de bytes para su trasmisión.

public static Object fromByteArray (byte [] bytes).

Debido a que la comunicación se realiza mediante socket UPD los objetos son recibidos son convertidos a un arreglo de bytes para su trasmisión este método lo carga como objeto.

public void registration(RegistrationEvent re).

Este método atiende el evento cuando un nuevo contenedor es registrado en el RegistrationMap, guarda la referencia a la dirección IP para realizar las notificaciones de la llegada de nuevos mensajes.

public void newMessages(QueueEvent qe) .

Este Método atiende el evento de nuevos mensajes, utiliza la lista de IP registradas para realizar un aviso multicast de que ha recibido nuevo mensaje, en la notificación incluye el identificador del destinatario para que los contenedores lo procesen y se conecte a

descarga el mensaje.

Interfaces

common.events.QueueListener

common.events.RegistrationListener

Server.DataStructure.AgentQueue

Este componente mantiene la información de los agentes que están dados de alta en el sistema e implementan las operaciones para acceder a las colas de mensajes de los agente.

Atributos.

private TreeMap binaryQueue.

Esta estructura mantiene la referencia entre los identificadores de los agentes y sus correspondientes colas de mensajes.

private Vector listeners.

Mantiene la referencia a los objetos manejadores del evento de la llegada de nuevos mensajes.

private static final int addNewAgent=0.

Esta constante se utiliza dentro del método binaryQueueAccess para determinar la operación que se va a realizar sobre la cola de mensajes del agente, esta constante indica que se va agregar un nuevo agente a la estructura binaryQueue.

private static final int getMsgQueue=1.

Esta constante se utiliza dentro del método binaryQueueAccess para determinar la operación que se va a realizar sobre la cola de mensajes del agente, esta constante indica

que se va obtener la referencia al queue de mensajes de un agente dado.

Métodos.

public AgentQueue().

Constructor de la clase

private synchronized Object binaryQueueAccess(int operation,Object agentID).

Debido a que el acceso a binaryQueue es de manera concurrente por varios hilos, el acceso se centraliza en este método para asegurar su integridad.

public void add(Object agentID).

Permite agregar un nuevo agente al binaryQueue.

public Object getQueue(Object agentID).

Obtiene la referencia a la cola de mensajes de un agente, regresa un valor nulo si el identificador no existe.

private Object remove(Object agentID).

Remueve un agente del binaryQueue.

public void addQueueListener(QueueListener ql).

Permite agregar un manejador de eventos de notificación de mensajes.

public void removeQueueListener(QueueListener ql).

Permite remover un manejador de eventos de notificación de mensajes.

synchronized public void notifyMsg(QueueEvent qe).

envía la notificación de nuevos mensajes a los manejadores de eventos que tiene registrados.

public synchronized boolean contain(Object ID).

Este método devuelve un valor verdadero en caso de que el binaryQueue contenga el identificador dado, en caso contrario regresa un valor de falso.

Server.DataStructure.MessageQueue

Este componente implementa la estructura para el control de la cola de mensajes, como el acceso es concurrente, implementa los controles necesarios para mantener la integridad y consistencia de la cola de mensajes.

Atributos.

private boolean flag.

Variable Global para uso interno de los métodos, la cual se utiliza para indicar cuando un proceso esta realizando una operación sobre la cola de mensajes.

private Vector shadow.

Su función es la de mantener un referencia a los mensajes que están siendo descargados por algún proceso, esta información se almacena ya que al terminarse estos mensajes serán eliminados de la cola principal, en caso de falla solo se elimina la referencias y no se afectan a la cola principal.

private Hashtable messages.

Guarda la referencia del mensaje asociada a su identificador, en la cola de mensajes de un agente. Actualmente el correcto funcionamiento de este componente recae en el programador ya que no hay ningún sistema que controle automáticamente el acceso a la

cola de mensajes, es responsabilidad del programador la correcta liberación del recurso

Métodos.

public MessageQueue().

Constructor de la clase.

synchronized private void setFlag() .

Cambia el valor de atributo flag, es usando cuando un proceso toma control de la cola de mensajes y cuando es liberada.

public void setShadow(Object ID).

Este método se utiliza para indicar el identificador de un mensaje que puede ser eliminado cuando se complete la transacción.

synchronized public void commit().

Cuando una transacción ha sido realizada exitosamente se elimina todos los mensajes que fueron indicados en la lista shadow.

synchronized public void rollback().

Cuando una transacción ha fallado o cuando se desea deshacer una operación se elimina las referencias indicadas en la lista shadow para que no sean eliminados los mensajes de manera permanente.

synchronized public void release().

Este método se utiliza para indicar cuando un proceso ha liberado la cola de mensajes

synchronized public boolean put(Wrapper msg).

Agrega un mensaje a la cola, internamente el proceso se encarga de conseguir el bloqueo

exclusivo y de liberar el recurso.

synchronized public Enumeration messages().

Este método puede representar mucho problemas, como regresa una referencia a la cola de mensajes, obtiene el bloqueo exclusivo sobre el recurso, pero no lo libera automáticamente, es responsabilidad del programador hacerlo , así como realizar el commit o el rollback. El mal uso de este método puede genera un bloqueo del recurso.

Server.DataStructure.GroupDefinition

Este componente contiene la información que define a un grupo en el sistema, debido a que el envío de mensajes para grupos se basa en la redistribución hacia cada uno de los miembros, este componente guarda la referencia del identificador de grupo y de los identificadores de los agentes que forma parte de el.

Atributos.

private Object id

Es el identificador del grupo.

private Vector members;

Guarda una lista de identificadores de agentes que pertenecen al grupo.

Métodos.

public GroupDefinition() .

Constructor de la clase.

public Object getId() .

Metodo para obtener el identificador del grupo.

public void setId(Object id) .

Metodo para colocar el identificador del grupo.

public Iterator getMembers() .

Obtiene una lista de los miembros del grupo.

public void addMember(Object member).

Agrega un identificador de un agente a la lista de miembros de un grupo.

public void delMember(Object member).

Elimina el identificador de un agente a la lista de miembros de un grupo.

public boolean isMember(Object member).

Regresa un valor verdadero si el agente es miembro de un grupo, de lo contrario regresa falso.

Server.DataStructure.GroupQueue

Debido a que no se maneja una cola de mensajes para cada grupo para evitar tener otra estructura que controlar, se definió que la cola de mensajes de grupo no es mas que una lista de referencias a los distintos grupos del sistema, así cuando un mensaje es enviado al grupo se obtiene una referencia a todos los miembros del grupo y se les entrega el mensaje en sus colas correspondientes, solo hay una sola copia del mensaje en memoria ya que las colas hacen referencia al mismo mensaje y este es eliminado cuando el ultimo miembro del grupo lo descarga ya que ningún objeto activo hace referencia a el.

Atributos.

private HashMap groupQueue.

Esta estructura mantiene la referencia entre los identificadores de los grupos y sus correspondientes definiciones.

private static final int addNewGroup=0.

Esta constate se utiliza para indica la operación que se va a realizar sobre groupQueue, en este caso es agregar un nuevo grupo.

private static final int getMsgQueue=1.

Esta constate se utiliza para indica la operación que se va a realizar sobre groupQueue, en este caso es obtener la referencia a la cola de mensajes del grupo, que esto se simula obteniendo la definición del grupo y así la lista de miembros a la cuales se les entregara el mensaje.

private static final int delGroup=2.

Esta constate se utiliza para indica la operación que se va a realizar sobre groupQueue, en este caso se elimina un nuevo grupo.

Métodos.

public GroupQueue() .

Constructor de la clase.

private synchronized Object QueueAccess(int operation,Object group).

debido a que el acceso al groupQueue es de manera concurrente, se implementa este método que controla el acceso al mismo para mantener la integridad y consistencia de la información. Las operaciones que controla son la de agregar y eliminar un grupo, además de obtener la referencia a su definición.

public Object getQueue(Object group).

Obtiene la referencia a la definición de un grupo.

public void add(Object group) .

agrega un nuevo grupo al groupQueue.

private Object remove(Object group).

remueve un grupo del groupQueue, actualmente no esta siendo utilizado ya que a carga se hace solo al iniciar el sistema y la estructura se lee del archivo SystemGroup.xml, para eliminar un grupo se debe eliminar su definición del archivo, al igual que para eliminar miembros del grupo pero se necesita reiniciar el servidor para cargar los cambios.

public Object getQueuesFrom(Object agentID) .

Obtiene una lista de los identificadores de grupo al cual pertenece un agente.

Server.DataStructure.RegistrationMap

El objetivo de este componente es el de guardar la información de los contenedores activos en el sistema, como lo es la dirección IP y el puerto de escucha para las notificaciones, una mejora necesaria para el servidor es que este componente realice monitoreo de los contenedores registrados para tener un mejor control de que contenedores siguen realmente activos.

Atributos.

private TreeMap binaryQueue.

Mantiene la referencia a la información de registro, la llave de busque es la IP del contendor.

private Vector listeners.

Mantiene una lista de componentes que atienden los eventos que lanzan cuando un nuevo contenedor se registra.

Métodos.

public RegistrationMap() .

Constructor de la clase.

public synchronized boolean add(RegistrationInformation ri).

agrega información de un contenedor.

public synchronized Object remove(Object ID).

elimina la información registra de un contenedor.

public void addRegistrationListener(RegistrationListener rl).

agrega una referencia al componente que manera los evento de registro de un contenedor.

public void removerRegistrationListener(RegistrationListener rl).

elimina la referencia a un componente que maneja los evento de registro de un contenedor.

4

Para ver el diagrama consulte los anexos.

Fig. 3.11 Diagrama de Clase AgentContainer

3.5 Descripción del paquete AgentContainer.

La funcionalidad principal del contenedor es la de brindar una ambiente de ejecución para los agentes, dentro del contenedor los agentes son ejecutados como hilos independientes cada uno realizando un operación diferente. Los agentes pueden agregarse al contenedor de manera dinámica en tiempo de ejecución, así que no es necesario detener el contenedor para poder agregar agentes, solo puede existir un contenedor por nodo el cual puede contener múltiples agentes interactuando entre si a través del paso de mensajes, en la figura 3.11 de muestra el diagrama de clase de este paquete.

3.5.1 Descripción de componentes de AgentContainer.

Dentro de los objetivos principales del contenedor, es la de implementar los mecanismo de comunicación para el envió y recepción de mensajes, además implementa una serie de mecanismo que simplifica el desarrollo de sistemas multiagentes., a continuación se realiza una descripción a detalle de las diversos componentes de este paquete así como las funcionalidades que brindan.

Client.Container

Su función principal es cargar todos los servicios necesarios para la ejecución de los agentes, implementa los esquemas de comunicación con el AMQServer para el paso de mensajes, define una serie de interfaces que se usan de referencia para comunicar los diferentes objetos en ejecución.

Atributos.

private Vector listeners.

Mantiene una lista de referencia a los manejadores de eventos del sistema.

private ContainerConnection cc.

Mantiene la referencia al componente que se enlaza con el servidor de mensajes, aunque

existan múltiples agentes en el sistema, solo se realiza una conexión para descargar los mensajes, internamente el contenedor administra el uso del canal de comunicación.

private EventAdapter adapter.

Mantiene la referencia al manejador de eventos del sistema.

private AgentLoader loader.

Hace referencia al componente que realiza la carga de agentes al contenedor.

private HashMap agents.

Mantiene una lista de los agentes que están cargados en el sistema.

private HashMap guisMap.

Mantiene una lista de los identificadores de los agentes y hacen referencia a clase de la interfaz grafica que tiene definida.

private Plugin plugin.

Hace referencia al componente que funciona como interfaz con el usuario para desplegar la interfaces de comunicación con los agentes disponibles en el contenedor.

Métodos.

public Container(String server, int port).

Constructor de la clase, se le pasa como parámetro la dirección IP, y el puerto de escucha del servidor de mensajes.

synchronized public void newMsg(QueueEvent qe).

Este método es lanzado por el manejador de eventos cuando recibe una notificación de un nuevo mensaje, se hace la validación de que el agente este registrado en el contenedor

de ser así se entabla conexión con el servidor para descargar el mensaje.

public void agentRegistration(Agent agent).

Este método se encarga de registrar a un agente en servidor de mensajes.

public void run().

Este método se ejecuta en un hilo independiente y cada minuto se conecta al servidor para revisar si hay mensajes para los agentes que tiene registrados.

public void send(Wrapper msg) .

Realiza el envió de mensajes a otros agentes, los agentes hace uso de este método mediante la interfaz AgentConnector.

public HashMap getGuisMap() .

Este método regresa la referencia a la estructura que almacena la información sobre los agentes y sus interfaces.

public static void main(String args[]).

A este método se le pasa como parámetro la IP del servidor de mensajes al cual se va a conectar, en caso de no pasarle ningún parámetro se conecta a localhost.

Interfaces.

ContainerInterface

Runnable

AgentConnector

Client.ContainerConnection

Este componente se encarga de realizar la conexión con el AQMServer para realizar el

envió y recepción de mensajes, este componente se queda cargado en memoria, mas no mantiene conexión con el servidor de manera permanente, cuando un agente envía un mensaje el contenedor entabla conexión con el servidor para realizar la transferencia del mensaje y se desconecta al terminar la transferencia, al igual cuando realiza la descarga de mensajes, recibe una referencia del agente del cual va a realizar la descargar de mensajes, durante esta operación solo un agente pude hacer uso de este componente, debido a que su estructura interna esta diseñada para descargar todos los mensajes de un agente a la vez. Actualmente este componente entabla conexión con el servidor cada vez que se realiza una operación, una forma de optimizar este esquema es manejar una cola operaciones a realizar y solo entablar una conexión.

Atributos.

private Socket coneccion.

Es el socket de conexión con el contenedor.

private ObjectInputStream entrada.

Es la referencia al Stream de entrada del enlace de comunicación.

private ObjectOutputStream salida

Es la referencia al Stream de salida del enlace de comunicación.

private int port.

Referencia al puerto de conexión del servidor

private int count.

Es el contador de mensajes que recibe un agente.

private String server.

Guarda la referencia a la dirección IP de servidor.

private boolean connected.

Se utiliza este atributo para indicar cuando hay conexión entablada con el servidor.

private Agent agent.

Hace referencia al agente que esta haciendo de la conexión para descargar los mensajes

Métodos.

public ContainerConnection(String server, int port).

Constructor de la clase, se le pasa como parámetro la dirección del servidor y puerto de conexión.

public void register().

Registra el contenedor en el servidor de mensajes.

private void conectarse().

Este método se encarga de entablar comunicación con el servidor.

public void run().

Este método se ejecuta para la recepción de mensajes por parte del servidor.

private void procesar (Wrapper msg).

Dependiendo del performative del mensaje invoca el método que lo procesara.

private void giveMsg(Wrapper msg).

Hace la entrega del mensaje a los queus de los agentes.

synchronized public void addAgent(AgentDefinition ad,String leaseTime).

Permite registrar un agente cargado en el contendor en el servidor de mensajes

synchronized public void requestMsg(Agent agent).

Este método permite solicitar los mensajes de un agente al servidor.

public void requestMsg(Iterator receivers).

Recibe una lista de agentes de los cuales descargar los mensajes del servidor.

private void processResponse(Wrapper msg).

Cuando el performative del mensaje es RESPONSE se procesa las respuesta recibidas por parte del servidor a una operación solicitada.

private void processCommand(Wrapper msg) .

Este método se realiza cuando el performative del mensaje es COMMAND, esto indica que el servidor ha enviando algún comando para ser ejecutado por el contenedor, actualmente no esta en uso.

synchronized private void send(Wrapper msg) .

Este método envía los mensajes al servidor a través del enlace de comunicación.

public void sendMsg(Wrapper msg).

Este método invoca al método send, la función principal es no dar acceso directo al método send, los agente envían los mensajes mediante la llamada de este método.

private void close().

Este método cierra la comunicación con el servidor.

Client.XMLParser

Este componente tiene la función de cargar los archivos de definición de agentes del sistema los cuales se encuentran en archivos con extensión xml , a continuación se describe el contenido del archivo:

Archivo de definición de agente.xml

```
<agent>
      <class>APP</class>
      <id>cid</id>
      <gui>CidGUI</gui>
</agent>
```

Tag	**Definición**
<agent>	Es el inicio de la definición del agente.
<class>	Es el nombre de la clase donde se implementa el agente.
<gui>	En caso de contar con un interfaz gráfica con este tag se define la clase que la implementa.

Atributos.

private SAXParser sparser.

Este atributo hace referencia a componente org.apache.xerces.parsers.SAXParser el cual se encarga de realizar el procesamiento de los archivos xml.

private Object agent.

Mantiene la referencia al objeto que se carga mediante el archivo xml que definen al agente dentro del contendor.

private StringBuffer value.

Variable Global para uso interno de los métodos.

private int set=0.

Variable Global para uso interno de los métodos.

private HashMap guisMap.

Guarda una lista con las referencia de los agentes que tiene definida un interfaz y guarda la referencia al objeto que implementa dicha interfaz.

Métodos.

public Iterator loadXML(int type).

Carga el archivo xml de configuración correspondiente dependiendo del tipo indicado.

public boolean readXML(String path).

Realiza la carga del archivo xml.

public void startDocument() .

Metodo sobre escrito de la clase DefaultHandler, que permite manejar cuando se inicia el archivo xml.

public void endDocument() .

Metodo sobre escrito de la clase DefaultHandler, que permite manejar cuando se llega al fin del archivo xml.

public void startElement(String uri, String localName, String qName, org.xml.sax.Attributes attributes).

Método sobre escrito de la clase DefaultHandler, que permite manejar cuando se lee un tag de inicio de elemento.

public void endElement(String uri,String localName,String rawName).

Método sobre escrito de la clase DefaultHandler, que permite manejar cuando se lee un tag de finalización de elemento.

public void characters (char ch[], int start, int length).

Método sobre escrito de la clase DefaultHandler, que permite obtener los valores de los elementos.

Herencia.

org.xml.sax.helpers.DefaultHandler

Client.AgentLoader

Este componente es el encargado de cargar los agentes dentro del contenedor; cada 10 segundo hace una revisión de la ruta donde se publican los agentes buscando nuevo archivos de definición

Atributos.

private String path.

Guarda la referencia a la ruta donde se publican los archivos de definición de los agentes.

private ListFilter filter.

Este elemento lo utiliza para realizar el filtrado de los archivos que va a cargar, archivos con extensiones xml.

private HashMap agents.

Este componente es el encargado de llenar esta estructura donde se guardan la referencia a los agentes dentro del contenedor.

private boolean ready.

Este atributo se utiliza para indicar cuando se puede realizar el acceso a la lista de agentes.

private AgentConnector container.

Es una referencia al contenedor, mediante la interfaz AgentConnector, la cual pasa como referencia a los agentes cuando los carga.

private HashMap guisMap.

Guarda una lista con las referencia de los agentes que tiene definida un interfaz y guarda la referencia al objeto que implementa dicha interfaz.

private XMLParser parser.

Hace referencia al componente que se encarga de realizar el procesamiento de los archivos xml para cargar los agentes.

Métodos.

public AgentLoader(String path,String mask,AgentConnector container,HashMap guisMap)

Es el constructor de la clase, se le pasan como parámetro la ruta de donde se van a leer los archivos de definición de agentes, la extensión de estos archivos; una referencia al contenedor mediante la interfaz AgentConnector, y una referencia a un HashMap donde se guardaran las referencias a las interfaces graficas de los agentes.

public void run() .

Este método se ejecuta en un hilo independiente y mediante el se realiza la carga del los agentes, además monitorea la ruta para cargar los nuevos agentes que se publiquen.

public HashMap getAgents() .

Devuelve una referencia a la estructura donde se almacena la información de los agentes que han sido cargados.

Interfaces.

Java.lang.Runnable

Client.EventAdapter

Este componente es el encargado de manejar los eventos remotos lazados por el servidor cuando hay un aviso de nuevo mensaje.

Atributos.

private ContainerInterface container.

Mantiene una referencia al contenedor para invocar el método newMsg cuando recibe la notificación de un mensaje nuevo.

Métodos.

public byte [] toByteArray(Object evt).

Debido a que la comunicación se realiza mediante socket UPD los objetos deben ser convertidos a un arreglo de bytes para su trasmisión.

public static Object fromByteArray (byte [] bytes).

Debido a que la comunicación se realiza mediante socket UPD los objetos son recibidos son convertidos a un arreglo de bytes para su trasmisión este método lo carga como

objeto.

public void run().

Este método se ejecuta en un hilo independiente ya que se queda en escucha de notificaciones remotas.

Interfaces.

Java.lang.Runnable

Client.GUI

Los Componentes de este paquete esta desarrollados con la finalidad de poder implementar interfaces gráficas para comunicarse con los agentes que están en ejecución en el contenedor, para realizar esta función cuenta con 3 componentes: Plugin, su función principal es brindar un medio de comunicación con el contenedor; AplicationLoader este componente presenta una lista de interfaces que puede ser cargadas, AgentFrame es una interfaz que debe implementar las interfaces gráficas que usen los agentes, a continuación un descripción detallada de estos componentes.

Client.GUI.Plugin

Este componente es una aplicación servidor que permite comunicarse con el contenedor y a través de ella podemos cargar la interfaz gráfica definidas en el contenedor.

Atributos.

private ContainerInterface container.

Mantiene la referencia al contenedor.

private Socket coneccion.

Es el socket de conexión con el contenedor.

private ObjectInputStream entrada.

Es la referencia al Stream de entrada del enlace de comunicación.

private ObjectOutputStream salida.

Es la referencia al Stream de salida del enlace de comunicación.

Métodos.

public Plugin(ContainerInterface container).

Constructor de la clase, se le pasa la referencia al contenedor.

public void run().

este método se ejecuta en un hilo independiente , este método implementa un servidor que se queda en escucha por el puerto 2111 el cual se utiliza para la comunicación con el contenedor.

public void setConnection(Socket clientSocket).

Este método se utiliza para configurar el enlace de comunicación.

public void processConnection().

Este método procesa los mensajes enviados por el AplicationLoader

public void procesar(Wrapper msg).

Este método procesa el mensaje en base al performative.

public void send(Wrapper msg).

Este método envía los mensajes

synchronized private void loadGUI(Wrapper msg).

Este método se utiliza para cargar las gui del agente, se recibe un mensaje y se realiza la carga.

private void sendGUIS(Wrapper msg) .

Este método envia una lista de las interfaces graficas disponibles.

public void close().

Cierra el enlace de comunicación con el servidor.

Client.GUI.AplicationLoader

Este componente brinda un listado de las interfaces graficas disponibles en el contenedor, permitiendo cargarla para comunicarse con los agentes.

Atributos.

private Socket coneccion.

Es el socket de conexión con el contenedor.

private ObjectInputStream entrada.

Es la referencia al Stream de entrada del enlace de comunicación.

private ObjectOutputStream salida.

Es la referencia al Stream de salida del enlace de comunicación.

private boolean connected.

Indica cuando se ha entablado la conexión.

Métodos.

public AplicationLoader() .

Constructor de la clase.

public void connect().

Entabla la conexión con el servidor, la dirección del servidor es localhost.

public void run().

Este método se ejecuta en un hilo independiente , este método se queda en espera de los mensajes enviados por el servidor.

synchronized public void send(Wrapper msg) .

Este método es el encargado de envía los Mensajes al contendor.

private void procesar(Wrapper msg) .

Procesa los mensajes en base a su performative, solo procesa los mensajes con performative RESPONSE.

private void processResponse(Wrapper msg).

Procesa el mensaje enviado por el servidor que es una lista de las interfaces graficas disponibles en el contenedor.

public void close().

Cierra el enlace de comunicación con el servidor.

public static void main(String args[]).

Metodo que inicia la ejecución del AplicationLoader no recibe parámetros de entrada.

Interfaces.

Java.lang.Runnable

Herencia.

javax.swing.JFrame

Client.GUI.AgentFrame

Es una interfaz que define los métodos que deben implementar las interfaces gráficas de los agentes para que puedan ser cargados por el contenedor.

public void setAgent(Object agent).

Mediante este método se le pasa como parámetro el agente con el cual se va a comunicar.

public void setVisible(boolean flag).

Este método hace visible la interfaz.

Para ver el diagrama consulte los anexos.

Fig. 3.12 Diagrama de Clase del paquete AMQCommon.

3.6 Descripción del Paquete AMQCommon

Varios componentes fueron agrupados dentro del paquete AMQCommon por su naturaleza, ya que son usados por ambos extremos tanto del lado cliente como del servidor.

3.6.1 Descripción de componentes del AMQCommon.

A continuación se hace una descripción detallada de los componentes.

Common.Wrapper

Este componente es utilizado para el envió de mensajes, se pude visualizar como un sobre donde se debe indicar el id, remitente, destinatario, prioridad, ontología, intención y el mensaje. El diseño de este componente se basa en la estructura definida de un mensaje del ACL KQML [31].

Atributos.

public static final int SUBSCRIBE=0.

Esta constante se utiliza para indicar el performative, del mensaje, se utiliza cuando un contenedor se va a registrar.

public static final int INFORM=1.

Esta constante se utiliza para indicar el performative, del mensaje, se utiliza cuando un mensaje es enviado a un grupo.

public static final int TELL=2.

Esta constante se utiliza para indicar el performative, del mensaje, se utiliza cuando un mensaje es enviado a un agente en particular.

public static final int REQUEST=3.

Esta constante se utiliza para indicar el performative, del mensaje, se utiliza cuando un agente solicita sus mensajes.

public static final int COMMAND=4.

Esta constante se utiliza para indicar el performative, del mensaje, se utiliza cuando un contenedor envía un comando al AMQServer, actualmente solo da de alta un agente.

public static final int RESPONSE=5.

Esta constante se utiliza para indicar el performative, del mensaje, se utiliza para las respuestas a los diversos mensajes.

private String id.

Es el identificador del mensaje. Cada vez que se instancia un objeto de esta clase se le asigna uno.

private String sender.

Es el identificador del remitente.

private String receiver.

Es el identificador del destino.

private String priority.

Se utiliza para manejo de prioridad de los mensajes, actualmente no implementado en l AMQServer.

private String ontology.

Se utiliza para indicar es el lenguaje de comunicación del mensaje, actualmente AMQServer lo implementa.

private int performative.

Se utiliza para indicar la intención del mensaje, hace uso de las constantes de clase definidas.

private Object obj.

Es el contenedor del mensaje, puede apunta a cualquier objeto serializable.

Métodos.

public Wrapper()

Constructor de la clase.

public String getPriority()

regresa la prioridad del mensaje

public String getId().

Obtiene el identificador del mensaje.

private void setId().

se utiliza para establecer el identificador del mensaje.

public int getPerformative() .

Obtiene la intención o performative del mensaje.

public void setPerformative(int performative) .

Se utiliza para establecer el performative o intención del mensaje.

public String getOntology().

Se utiliza para obtener la ontología del mensaje.

public void setOntology(String ontology) .

Se utiliza para establecer la ontología del mensaje.

public void setPriority(String priority) .

Se utiliza para establecer la prioridad del mensaje.

public Object getObj().

Se utiliza para obtener el objeto del mensaje.

public void setObj(Object obj).

Se utiliza para establecer el objeto del mensaje.

public String getReceiver().

se utiliza apara obtener el destinatario de mensaje.

public void setReceiver(String receiver) .

se utiliza apara establecer el destinatario de mensaje.

public String getSender() .

se utiliza apara obtener el remitente de mensaje.

public void setSender(String sender).

se utiliza apara establecer el remitente de mensaje.

Interfaces.

public class RegistrationInformation implements Serializable

Common.RegistrationInformation

Este componente se utiliza para generar la información de registro que el contendor le envia al servidor de mensajes.

Atributos.

public static final int FOREVER=0.

Esta constante se utiliza para indicar que el registro es por tiempo indeterminado, actualmente el registro es de manera indefinida.

private InetAddress ip.

dirección IP del contenedor.

private int port.

puerto UDP donde va estar escuchando el contenedor las notificaciones remotas

private int leaseTime.

tiempo de registro del contenedor, actualmente el registro es permanente, se necesita implementar el esquema para registrarlo por tiempo determinado.

Métodos.

public RegistrationInformation() .

Constructor de la clase.

public InetAddress getIp() .

Permite obtener la IP del contendor.

public void setIp(InetAddress ip) .

Establece la ip del contenedor.

public int getLeaseTime() .

Obtiene el tiempo de registro del contenedor.

public void setLeaseTime(int leaseTime).

Establece el tiempo de registro del contenedor.

public int getPort().

Obtiene el puerto UDP de escucha para las notificaciones remotas.

public void setPort(int port).

Establece el puerto UDP de escucha para las notificaciones remotas.

Interfaces.

java.io.Serializable

Common.Agent.Agent

Esta clase abstracta define los métodos mínimos que debe definir el programador para implementar un agente, que son newMessages que se ejecuta cuando se recibe un mensaje y el método behaviour que es lo que realiza el agente cuando se carga en memoria, quedando a libertad del programador la implementación de estos métodos; esto permite centrarse en la definición del comportamiento del agente. El contenedor implementa el los procedimientos para comunicarse con otros agentes.

Atributos.

private AgentConnector container.

Mantiene referencia al contenedor, mediante el uso de esta referencia el agente envía los mensajes.

private Object ID.

Es el identificador del agente.

private Stack queue.

Es la cola de mensajes del agente.

private boolean set.

Indica cuando al agente se le ha establecido un identificador. Una vez que se ha establecido un identificador no se puede cambiar.

Métodos.

public Agent() .

Constructor de la clase.

final public void setID(Object ID) .

Establece el identificador del agente.

final public Object getID().

Obtiene el identificador del agente.

public void setContainer(AgentConnector container) .

Establece la referencia al contenedor, el agente hace uso de esta referencia para el envió de mensajes.

final public void giveMsg(Wrapper msg).

Entrega el mensaje en la cola de mensajes del agente, este método es ejecutado por el contenedor cuando descarga los mensajes del servidor.

public Stack getQueue() .

Obtiene el queue de mensajes del agente.

final public void send(String receiver,Object contain).

Mediante este método el agente envia los mensajes en esquema uno a uno.

final public void publish(String group,Object contain).

Mediante este método el agente envia los mensajes en esquema uno a muchos.

final public void run().

Este método se ejecuta en un hilo independiente y ejecuta el método behaviour

final public void start().

Inicia la ejecución del método run.

public abstract void behaviour().

Este método debe ser implementado por la clase que hereda de este, aquí se define el comportamiento del agente.

public abstract void newMessages().

Este método debe ser implementado por la clase que hereda de este, Este método se ejecuta por el contendor para notificar cuando ha llegado un mensaje, se debe indicar que hacer cuando ocurre esto.

Interfaces.

Java.lang.Runnable

Common.agent.AgentDefinition.

Es te componente lo utiliza el servidor de mensajes para cargar en memoria la definición del agente.

Atributos.

private Object id.

Es el identificador del agente.

private Object cert.

La finalidad es contar con un certificado de seguridad para autentificar a los agentes actualmente no esta implementada esta autentificación

private Object leaseTime.

La finalidad es designar un tiempo de registro al agente, no esta implementado.

Métodos.

public AgentDefinition() .

Constructor de la clase sin parámetros.

public AgentDefinition(Object id, Object cert, Object leaseTime).

Constructor de la clase, se le pasa el identificador, el certificado y el tiempo de registro como parámetros.

public Object getCert() .

Obtiene el certificado del agente.

public void setCert(Object cert) .

Establece el certificado del agente.

public Object getId().

Obtiene el id del agente.

public void setId(Object id).

Establece el id del agente.

public Object getleaseTime().

Obtiene el tiempo de registro del agente.

public void setleaseTime(Object leaseTime).

Establece el tiempo de registro del agente

Interfaces.

java.io.Serializable

Common.Agent.AgentConnector

Mediante esta interfaz el agente hace el envió de mensajes, el contenedor implementa esta interfaz simplificando el envió de mensajes para el agente.

common.command

Este componente se utiliza para enviar comando, la estructura interna esta definida para manejar el comando mediante constantes de clase, adema define un arreglo para los parámetros y una referencia al target del comando.

Atributos.

public static int ADDAGENT=0.

Este constante se utiliza para cuando se va dar de alta un agente en el servidor de mensajes.

public static int DELAGENT=1.

Este constante se utiliza para cuando se va dar de baja un agente en el servidor de mensajes, actualmente el servidor no reconoce este comando.

int command.

Mediante Este atributo se define el comando a ejecutar.

String[] params.

Maneja la lista de parámetros de comando.

Object target

Es el objeto sobre el cual se va ejecutar el comando, debe ser una clase que implemente la interfaz serializable.

Métodos.

public Command(int command, String[] params,Object target) .

Constructor de la clase que recibe el comando, la lista de parámetros y el target

public Command(int command,Object target).

Constructor de la clase , recibe el comando sin parámetros.

public void setParams(String[] params).

Establece la lista de parámetros del comando.

public String[] getParams().

Establece la lista de comandos.

public void setCommand(int command).

Establece el comando.

public int getCommand().

Obtiene el comando.

public Object getTarget().

Obtiene el target del comando.

public void setTarget(Object target).

Establece el target del comando.

Interfaces.

java.io.Serializable

common.events.QueueEvent

Se utiliza para el envió de eventos de las colas de mensajes.

Atributos.

private String msg.

Se pasa como parámetro el id de agente que ha recibido el mensaje.

Métodos.

public QueueEvent(Object source, String msg) .

Constructor de la clase.

public String getMsg() .

Obtiene el mensaje del evento.

public void setMsg(String msg) .

Establece el mensaje del evento.

Interfaces.

java.io.Serializable

Herencia.

java.util.EventObject

common.events.QueueListener

Esta interfaz implementa el método newMessages que se ejecuta cuando se recibe una notificación de nuevo mensajes.

common.events.RegistrationEvent

Este componente se utiliza para enviar la información de registro cuando se envia una notificación.

Atributos.

RegistrationInformation registrationInformation

Mantiene la información de registro que disparo el evento.

Métodos.

public RegistrationEvent(Object source, RegistrationInformation registrationInformation).

Constructor de la clase.

public RegistrationInformation getRegistrationInformation().

Obtiene la información de registro que disparo el evento.

public void setRegistrationInformation(RegistrationInformation registrationInformation).

Establece la información de registro que género el evento.

Interfaces.

java.io.Serializable

Herencia.

java.util.EventObject

Capitulo IV

Demostración del uso del Framework.

4.1 Implementando nuestro primer agente.

Para implementar un agente se debe extender de la clase common.agent.Agent dentro del jar AMQCommon; la cual es una clase abstracta de la cual debemos implementar los siguientes métodos.

```
public abstract void behaviour();
public abstract void newMessages();
```

Como programadores es nuestra responsabilidad implementar el método bahaviour() según las tareas que vaya a realiza el agente, y en el método newMessages() se define el comportamiento del agente cuando reciba algún mensaje como se muestra en el siguiente código.

```
import common.Wrapper;
import common.agent.Agent;

public class HelloWorldAgent extends Agent

{
    @Override
    public void behaviour()
    {
        try
        {
            Thread.sleep(30000);
            send("cid", "hello world");
        }
        catch (InterruptedException ex){  }
    }

    @Override
    public void newMessages()
```

```
    {
       System.out.println("new messages");
       Stack queue=getQueue();
       while(!queue.empty())
       {
            Wrapper msg=(Wrapper) queue.pop();
            System.out.println("from>>"+msg.getSender()
                  +">>to>>"+msg.getReceiver()+">>"+msg.getObj());
       }
    }
}
```

4.2 Iniciar ejecución del agente.

Todos los agentes corren dentro de un entorno de ejecución denominado AgentContainer, el cual brinda todas las funcionalidades al agente para el envió y recepción de mensajes, simplificando estas tareas para el programador. Después de implementar el agente debemos realizar 2 pasos para ponerlo en funcionamiento, publicar el agente e iniciar el Contenedor

4.2.1 Publicación del agente dentro del Framework.

El AgentContainer carga en memoria e inicia la ejecución de los agentes, esto lo hace a través del archivo de definición del agente en formato xml, como el que se muestra a continuación.

```
<agent>
      <class>HelloWorldAgent</class>
      <id>cid</id>
</agent>
```

Tag	Descripción
<agent>	marca el inicio de la definición del agente.
<class>	indica la clase que implementa al agente(esta tag siempre debe ser definida primero).
<id>	Se utiliza para definir el id con el cual se cargara el agente dentro del sistema.
<gui>	esta tag es utilizada para definir si el agente cuenta con interfaz grafica para interactuar con el usuario, su funcionamiento será mostrado posteriormente.

Para publicar el agente es necesario colocar el archivo xml dentro de la carpeta publish la cual se encuentra localizada en el path a la altura del paquete jar del AgentContainer. El contenedor carga los archivos de definición en tiempo de ejecución, permitiendo agregar agentes al sistema sin necesidad de detener el contenedor.

4.2.2 Iniciar el contenedor.

Durante el arranque de contenedor tenemos varias opciones que nos permiten modificar su comportamiento por default; mediante la opción -RemotePath le indicamos la ruta de donde se encuentran las clases que implementan a los agentes, la rutas deben ser definidas en forma de URL y separadas por ","; de no definirse esta opción cargara las clases de forma local de "FILE:Classes/"; mediante la opción –AMQServer se puede indicar la dirección del servidor de mensajes al cual nos vamos a conectar, en caso de no indicarse se conectara a localhost; la opción -CTime se utiliza para definir los periodos de conexión del contenedor al servidor de mensaje, si no es indicada por default el periodo de conexión será de 60000 milisegundos. A continuación se muestra el ejemplo de como iniciar el AgentContainer.

```
cd $ACONTAINER_PATH
java  -jar AgentContainer.jar -AMQServer AMQS_IP_address
-RemotePath File:/usr/local/classes/,http://ip_address/Classes/
-CTime 60000
```

4.3 Implementado un Agente con GUI.

El framework nos permite definir interfaces graficas de usuario (GUI) para comunicarnos con los agentes, mediante esto se busca facilitar el desarrollo de aplicaciones de frontend que realicen su operaciones mediante agentes, el contenedor incluye un plugin mediante el cual nos permite listar las aplicaciones graficas que se encuentran disponible para su uso, como se muestra en la siguiente figura.

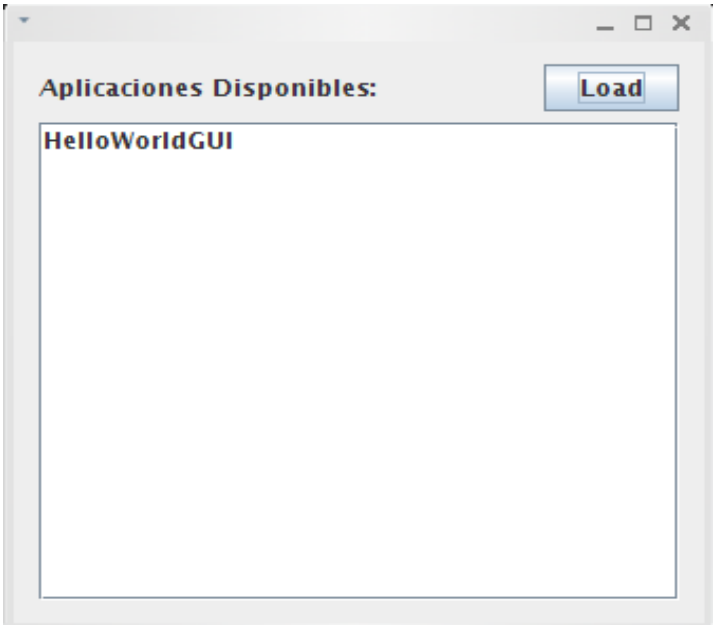

Fig. 4.1 ApplicationLoader

El ApplicationLoader nos despliega las GUIs disponibles en nuestro contenedor, y a través de este podemos cargar la aplicación con tan solo seleccionarla de la lista y presionar el botón load.

Para iniciar la ejecución del ApplicationLoader debemos realizar lo siguiente.

```
cd $ACONTAINER_PATH
java -cp AgentContainer.jar Client.GUI.AplicationLoader
```

4.3.1 Desarrollar una Aplicación grafica para nuestro agente.

Para implementar un interfaz grafica para comunicarnos con el agente, la clase debe de implementar la interfaz AgentFrame, la cual define los siguientes métodos.

```
public void setAgent(Object agent);
public void setVisible(boolean flag);
```

El método setAgent(Object agent) es utilizado por el contendor para pasar la referencia del agente a la clase que implementa la GUI cuando es invocado por el ApplicationLoader. Es responsabilidad del programador guardar una referencia al agente que le es pasado como argumento, ya que es el canal de comunicación entre el agente y la GUI. El método setVisible(boolean flag) es utilizado por el contendor para desplegar la aplicación grafica referenciada al agente.

Para el siguiente ejemplo modificaremos nuestra clase HelloWorldAgent para que despliegue su mensaje dentro de un JFrame.

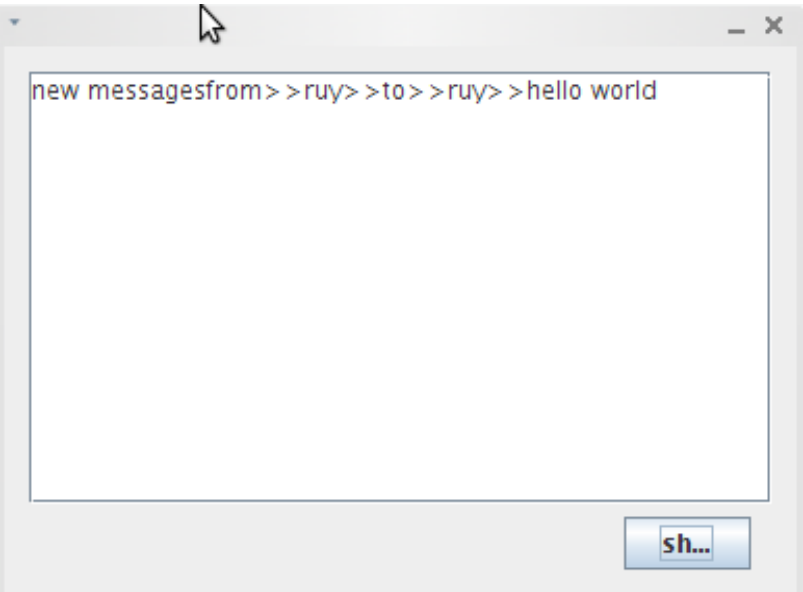

Fig. 4.2 Aplicación grafica para desplegar los mensajes del HelloWorldAgent.

Código HelloWorldGUI que define la interfaz grafica para HelloworldAgent.

```
import Client.GUI.AgentFrame;

private javax.swing.JButton jButton1;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JTextArea jTextArea1;

public class HelloWorldGUI extends javax.swing.JFrame implements
AgentFrame
{
      private HelloWorldAgent agent;

      public HelloWorldGUI()
      {
                  initComponents();
      }
```

```
        public void setAgent(Object agent)
        {
                    this.agent=(HelloWorldAgent2) agent;
                    this.agent.setOut(jTextArea1);
        }

        private void initComponents()
        {
              //inicializacion de los componenetes graficos
              ...........
              setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        }

        private void winlistener(java.awt.event.WindowEvent evt)
        {
                     if(evt.getNewState()==0)
                    {
                          this.agent.unsetOut();
                          System.out.println("unset HelloWorldGUI
                          from agent: "+this.agent.getID());
                    }
         }

        private void
        jButton1ActionPerformed(java.awt.event.ActionEvent
                                    evt)
        {
              this.agent.newMessages();
        }
}
```

Como se ve en el código anterior dentro del método setAgent guardamos la referencia al agente, un punto a remarcar es que el contenedor nos pasa la referencia del agente en una clase Object, así que debemos realizar el casting a la clase especifica a utilizar; en este

ejemplo dentro de este método estamos pasando al agente el componente grafico donde enviara la salida por lo cual será nuestra responsabilidad eliminar esta referencia cuando cerremos la aplicación. El método setVisible es implementado por la clase JFrame pero en caso de extender de una clase que no lo implemente será nuestra responsabilidad definirlo. Un punto importante para usar las interfaces graficas sin afectar el funcionamiento del contenedor es poner la operación por default de cierre del JFrame en DISPOSE en lugar de EXIT, de lo contrario cerraría el contenedor.

```
setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
```

4.3.2 Publicar el agente con su interfaz gráfica.

Internamente El AgentContainer guarda una referencia a la clase que implementa la GUI del agente, el nombre de la clase se encuentra en el archivo de definición xml indicada por la tag <gui>, como se muestra a continuación.

```
<agent>
      <class>HelloWorldAgent</class>
      <id>ruy</id>
      <gui>HelloWorldGUI</gui>
</agent>
```

El contendor internamente mantiene un referencia a la clase definida dentro de la tag <gui>, la cual se cargara cuando sea solicitada por el ApplicationLoader. El contenedor buscara las clases que implementan las GUIs en las misma rutas donde carga los agente, si tenemos algún Path diferente para estas clases hay que definirlas en la opción -RemotePath al iniciar el AgentContainer.

4.4 Iniciando el AMQServer

Para hacer uso del framework requerimos iniciar el AMQServer, para iniciar su ejecución

debemos realizar lo siguiente.

```
cd $AMQSERVER_PATH
java -jar AMQServer.jar
```

4.5 Archivo de definición de agentes en el AMQServer.

El sistema mantiene una relación de los agentes dados de alta en el sistema, los cuales son registrados de forma automática por los contenedores cuando son publicados, por lo cual el identificador debe ser único para todo el sistema, en caso del que el identificador se encuentre repetido no se realizara la carga del agente dentro del contenedor, el archivo donde están registrados los agentes del sistema se encuentra en la siguiente ruta $AMQSERVER_PATH/config/ bajo el nombre de SystemAgents.xml el cual contiene el siguiente formato.

```
<SystemAgents>
<Agent>
      <ID>cid</ID>
      <Cert>231182</Cert>
      <LeaseTime></LeaseTime>
</Agent>
</SystemAgents>
```

Tag	**Descripción.**
<SystemAgents>	marca el inicio de la lista de agentes dentro del sistema.
<Agent>	marca el inicio de la definición del agente.
<ID>	es el identificador único del agente dentro del sistema.
<Cert>	no implementada, planeada para ser utilizada en la validación de la comunicación entre agentes mediante el uso de certificados
<LeaseTime>	no implementada, se planeo para registrar los agentes por periodos de tiempo definidos.

4.6 Definición de Grupos.

El sistema permite el envió de mensajes a grupos, mediante el cual podemos hacer listas de agentes para el envió de mensajes masivos, el sistema carga la definición de esos grupos del archivo $AMQSERVER_PATH/config/SystemGroups.xml, el cual tiene el siguiente formato.

```
<SystemGroups>
    <Group>
        <ID>ALL</ID>
        <Members>
            <Agent>ALL</Agent>
        </Members>
    </Group>
    <Group>
        <ID>TEST</ID>
        <Members>
            <Agent>cid</Agent>
            <Agent>tania</Agent>
        </Members>
    </Group>
</SystemGroups>
```

Tag	Descripción.
<SystemGroups>	define el inicio del listado de los grupos.
<Group>	indica el inicio de la definición de grupo.
<ID>	el cual define el identificador único de grupo, este identificador debe ser diferente a cualquier identificador utilizado por algún agente.
<Members>	mostrara la lista de agentes que pertenecen al grupo.
<Agent>	Indica los identificadores de los agentes que pertenecen al grupo.

4.7 Sistema de administración de redes basado en agentes.

Para demostrar una aplicación mas avanzada del uso del framework y de la factibilidad del desarrollo de aplicaciones con metodología de agentes, se desarrollo el siguiente sistema de control para acceso a los recursos dentro de una red mediante la autentificación del usuario vía agentes. El sistema se divide en 2 dos componentes principales el sistema de Control acceso a los recurso y el cliente de autentificación como se muestra en la siguiente figura.

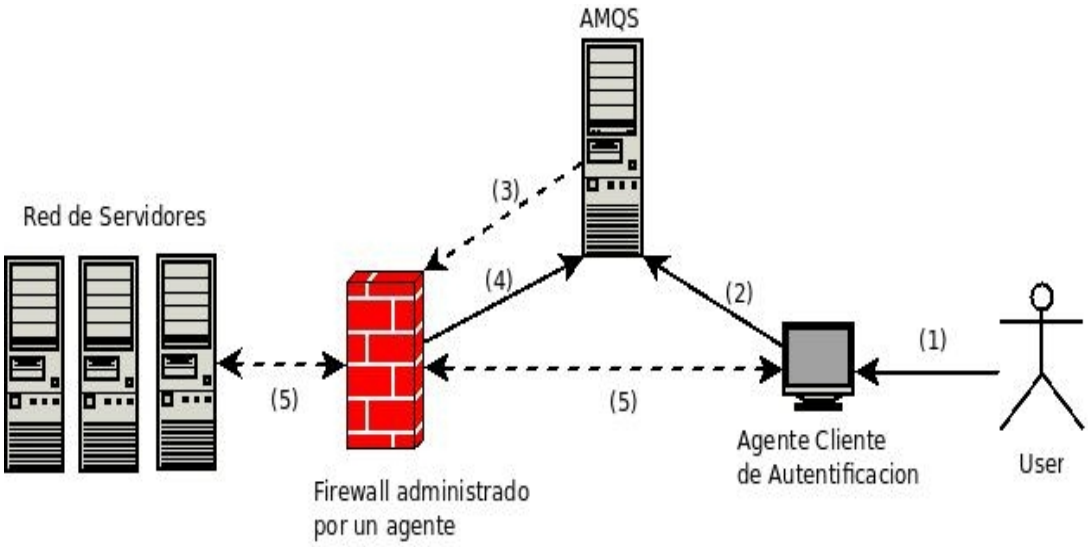

Fig. 4.3 Esquema de funcionamiento del sistema de administración de red basado en agentes.

Descripción de funcionamiento del sistema.

1.-El usuario se autentifica en la aplicación cliente.

2.-El agente cliente envía un mensaje al agente encargado de la administración del firewall con el nombre del usuario, el password y la IP del equipo donde esta localizado el usuario.

3. El servidor de mensajes (AMQS) envía una notificación de recepción de nuevo mensaje.

4.- El agente localizado en el firewall descarga y procesa el mensaje, durante esta fase se valida al usuario y determinar las políticas de acceso que tiene asignados el usuario y se modifica las reglas del firewall.
5.- El equipo cliente tiene acceso a los servicios asignados.

4.7.1 Sistema de Control de acceso a los recursos.

Para poder llevar acabo la restricción a los servicios de una red de manera transparente sin realizar ninguna modificación a los sistemas actuales, se opto por filtrar las tramas ethernet utilizando un firewall con sistema operativo Linux y un agente encargado de modificar las configuraciones del firewall para darle los accesos a los usuarios. El esquema es simple, delegamos al agente las tareas que normalmente realizaría el administrador de la red para administrar firewall.

Bajo este esquema el acceso a los recursos se hace de manera transparente para el usuario, sin importar desde que terminal ingrese a la red; los agentes serán los encargados de realizar las configuraciones necesarias para darles acceso. A continuación se describirán cada uno de los componentes requeridos para la implementación del sistema.

4.7.2 Firewall en Linux con iptables

El núcleo de este sistema son las IPtables, el cual es un sistema de firewall vinculado al kernel Linux que se ha extendido enormemente a partir del kernel 2.4. Para su funcionamiento es necesario aplicar la reglas de filtrado, mediante el uso del comando iptables, el cual nos permite añadir, borrar y crear reglas. Basándonos en esta característica nuestro agente armara las reglas en base a la información enviada por el agente cliente y en las políticas definidas en el sistemas. A continuación se muestra las reglas utilizadas por el sistema.

```
//habilitamos el reenvió de paquetes
1:    echo 1 > /proc/sys/net/ipv4/ip_forward
//limpiar las iptables
2:    iptables -F
3:    iptables -Z
4:    iptables -X
5:    iptables -t nat -F
//política por default drop
6:    iptables -P INPUT DROP
7:    iptables -P OUTPUT DROP
8:    iptables -P FORWARD DROP
//acceso a la interfaz de conexión al área de los servidores
9:    iptables -D INPUT -i wlan0 -j ACCEPT
10:   iptables -D OUTPUT -o wlan0 -j ACCEPT
//acceso al AMQS
11:   iptables -A INPUT -s 192.168.23.0/24 -p tcp --dport 2311 -j
ACCEPT
12:   iptables -A OUTPUT -d 192.168.23.0/24 -p tcp --sport 2311 -j
ACCEPT
//habilitar envió y recepción de paquetes por la ip 192.168.23.254
13:   iptables -A INPUT -s 192.168.23.254 -j ACCEPT
14:   iptables -A OUTPUT -d 192.168.23.254 -j ACCEPT
//acceso para recepción de eventos remotos del AgentContainer
15:   iptables -A INPUT -s 192.168.23.0/24 -p udp  -j ACCEPT
16:   iptables -A OUTPUT -d 192.168.23.0/24 -p udp -j ACCEPT
```

Para este ejemplo se utilizo política drop por default, por lo cual se debe definir explícitamente, la reglas de entrada y de salida para las tramas. Para mayor referencia sobre el uso de iptables visite: http://www.pello.info/filez/firewall/iptables.html.

4.7.3 Agent Network Management System.

Son los componentes java desarrollados para el funcionamiento del sistema los cuales se listan a continuación:

ANMS(Agent Network Management System).

Este agente es el responsable de cargar las reglas de los usuarios las cuales se encuentra definidas dentro de archivos en formato XML. Las definiciones son cargadas y almacenadas en memoria para ser aplicadas cuando un usuario se autentifique en el sistema.

La ruta en donde se almacenan la reglas es la siguiente /usr/local/ANMSystem/rules, el ANMS revisa cada minuto la ruta para cargar o modificar las reglas almacenadas en memoria.

El ANMS recibe los mensajes de los clientes con los datos de usuario, password y dirección IP, con esto confirma la identidad del usuario y arma las reglas del iptables con la información almacenada en memoria.

Código ANMS.java

```
public class ANMS extends Agent
{
   private String path="/usr/local/ANMSystem/rules";
   private File[] list;
   private HashMap listMap=new HashMap();
   private HashMap rulesMap=new HashMap(),ipchains=new HashMap();
   private RuleDefinition rl;
   private ListFilter filter=new ListFilter("xml");
   private boolean flag=false;

   private void loadRules()
   {
        if (path==null) throw new Error("Path no set");
        File folder=new File(path);
```

```
        if(!folder.isDirectory()) throw new Error("Folder not
exist");
        if(!folder.exists()) throw new Error("Folder not exist");
        XMLParser parser=new XMLParser();
        list = folder.listFiles(filter);
        for (int x = 0; x < list.length; x++)
        {
             listMap.put(list[x].getName(),
list[x].lastModified());
             rl=(RuleDefinition)
parser.loadXML(path,list[x].getName());
             rulesMap.put(rl.getID(), rl);
             System.out.println(list[x].getName() +" loaded" );
        }
        flag=true;
        while(true)
        {
                try
                {
                    Thread.sleep(60000);
                }
                catch (InterruptedException ex){}
                list = folder.listFiles(filter);
                for (int x = 0; x < list.length; x++)
                {
                    Long
                  value=(Long)listMap.get(list[x].getName());
                    long modified=value!=null?value.longValue():0;
                    long lm=list[x].lastModified();
                  if(modified==0 || modified!=lm)
                  {
                rl=(RuleDefinition)parser.loadXML(path,list[x].get
                  Name());
                       rulesMap.put(rl.getID(), rl);
                       System.out.println(list[x].getName() +"
```

```
loaded" );
                    }
                }
                listMap=new HashMap();
                for (int x = 0; x < list.length; x++)
                {
                    listMap.put(list[x].getName(),
list[x].lastModified());
                }
                System.gc();
            }
   }

   public void applyRulesTo(String ID,String pass,String ip)
   {
        while(!flag)
        {
            try
            {
                Thread.sleep(1000);
            }
            catch (InterruptedException ex{}
        }
        System.out.println("applyRulesTo "+ID);
        Iterator rules;
        RuleDefinition rules2apply=(RuleDefinition)
rulesMap.get(ID);
        if(rules2apply!=null && rules2apply.checkPassword(pass))
        {
            rules=rules2apply.getServices();
            while(rules.hasNext())
            {
                ServiceDefinition service=(ServiceDefinition)
                  rules.next();
                String cmd="iptables -A FORWARD ";
```

```
                if(!service.getDestination().isEmpty())
                 cmd+="-s "+ip+" -d "+service.getDestination()+"
                  ";
                if (!service.getProtocol().isEmpty())
                {
                    cmd+="-p "+service.getProtocol()+" ";
                    if(!service.getPort().isEmpty())
                        cmd+="--dport "+service.getPort()+" ";
                }
                cmd+="-j ACCEPT";
                System.out.println(cmd);
                try
                {
                    String old_User=(String) ipchains.get(ip);
                    if(old_User!=null) unsetRulesTo(old_User,ip);
                    Runtime.getRuntime().exec(cmd);
                    ipchains.put(ip, ID);
                }
                catch (IOException ex)
                {
                    Logger.getLogger(ANMS.class.getName()).log(Lev
                        el.SEVERE, null, ex);
                }
            }
        }
        else
            System.out.println("user undefined or wrong password
            "+ID);
    }

    @Override
    public void behaviour()
    {
        loadRules();
    }
```

```
    @Override
    public void newMessages()
    {
            Stack queue=getQueue();
            while(!queue.empty())
            {
                Wrapper msg=(Wrapper) queue.pop();
                System.out.println("from>>"+msg.getSender()
+">>to>>"+msg.getReceiver()+">>"+msg.getObj());
                String
tokens[]=msg.getObj().toString().split(":");
                applyRulesTo(tokens[0],tokens[1],tokens[2]);
            }
    }

    private void unsetRulesTo(String old_User,String ip)
    {
        System.out.println("unset Rules To "+old_User);
        Iterator rules;
        RuleDefinition rules2apply=(RuleDefinition)
              rulesMap.get(old_User);
        if(rules2apply!=null)
        {
            rules=rules2apply.getServices();
            while(rules.hasNext())
            {
                ServiceDefinition service=(ServiceDefinition)
                rules.next();
                String cmd="iptables -D FORWARD ";
                if(!service.getDestination().isEmpty())
                 cmd+="-s "+ip+" -d "+service.getDestination()+"
";
                if (!service.getProtocol().isEmpty())
                {
```

```
                    cmd+="-p "+service.getProtocol()+" ";
                    if(!service.getPort().isEmpty())
                        cmd+="--dport "+service.getPort()+" ";
                }
                cmd+="-j ACCEPT";
                System.out.println(cmd);
                try
                {
                    Runtime.getRuntime().exec(cmd);
                }
                catch (IOException ex){}
            }
        }
    }
}
```

El método load rules es el encargado de realizar la carga y actualizar la reglas para el iptable en el sistema, como se muestra en el código cada minuto se revisa el contenido de la carpeta ANMSystem/rules revisando si algún archivo ha cambiado. El formato del archivo de definicion de reglas tiene el siguiente formato:

```
<user>
      <id>rodrigo.dominguez</id>
      <pass>9k47Gh</pass>
      <services-list>
            <service>
                  <destination>192.168.1.1</destination>
                  <port>80</port>
                  <protocol>tcp</protocol>
            </service>
      </services-list>
</user>
```

Tag	Descripcion.
<user>	marca el inicio de la definicion de un usuario.
<id>	Es el identificador con el cual el usuario esta dado de alta dentro del sistema.
<pass>	Es el password contra el cual se va a validar el usuario.
<services-list>	Indica el inicio de definicion de la lista de servicios que tiene dado de alta el usuario.
<service>	Define el inicio de la descripcion del servicio.
<destination>	IP al cual el usuario va tener acceso.
<port>	Numero puerto por el cual esta escuchando la aplicacion a la cual se va conectar el usuario.
<protocol>	Tipo de protocolo que usa la aplicacion (TCP/UDP).

Por cada usuario dado de alta en el sistema se debera definir un archivo como el anterior, es importante definir todo los elemento indicados ya que son requerido para la correcta construccion de la reglas del firewall.

El método applyRulesTo lo ejecuta el agente ANMS cuando recibe un mensaje de algún cliente para aplicar las reglas en el firewall. El método unsetRulesTo se ejecuta para remover las reglas aplicadas para un host antes de aplicar las nuevas.

RuleDefinition

Para cada usuario se almacena en memoria la información de su ID, password y reglas de acceso a los recursos, dentro de la clase RuleDefinition la cual implementa los métodos para inicializar y verificar los datos.

Código RuleDefinition.java

```
public class RuleDefinition
```

```
{
   private String ID,password="";
   private Vector services;

    public RuleDefinition(String ID)
    {
        this.ID=ID;
        this.services=new Vector(1,1);
    }

    public void setPassword(String password)
    {
        this.password = password;
    }

    public boolean checkPassword(String password)
    {
        return this.password.equals(password);
    }

    public String getID()
    {
        return ID;
    }

    public Iterator getServices()
    {
        return services.iterator();
    }

    public int setServices(ServiceDefinition services)
    {
        this.services.add(services);
        return 0;
    }
```

```
    public int countServices()
    {
        return this.services.size();
    }
}
```

ServiceDefinition

Esta clase almacena la información necesaria para construir las reglas del firewall para los accesos permitidos de los usuarios, como la IP, el puerto y protocolo que utiliza la aplicación. Por cada regla definida se crea un objeto de este tipo que es almacenado por la clase RuleDefinition.

Código ServiceDefinition.java

```
public class ServiceDefinition
{
    private String destination;
    private String port;
    private String protocol;

    public String getDestination()
    {
        return destination;
    }

    public void setDestination(String destination)
    {
        this.destination = destination;
    }

    public String getPort()
    {
```

```
        return port;
    }

    public void setPort(String port)
    {
        this.port = port;
    }

    public String getProtocol()
    {
        return protocol;
    }

    public void setProtocol(String protocol)
    {
        this.protocol = protocol;
    }

    public int compareTo(ServiceDefinition service)
    {
        if(!this.destination.equals(service.getDestination()) )
          return -1;
        if(!this.port.equals(service.getPort()))
            return -1;
        if(!this.protocol.equals(service.getProtocol()))
            return -1;
        return 0;
    }
}
```

XMLParser.

Debido a que las reglas se encuentran dentro de archivos con formato XML se utiliza un parser para leer los datos y crear los objetos RuleDefinition.

Código XMLParser.java

```
public class XMLParser extends DefaultHandler
{
    private SAXParser sparser;
    private StringBuffer value;
    private int set=0;
    private RuleDefinition rl;
    private ServiceDefinition sd;

    public Object loadXML(String path,String file)
    {
        if ( readXML(path + File.separator + file) )
            return rl;
        else
            return null;
    }

    public boolean readXML(String file)
    {
        sparser=new SAXParser();
        try
        {
            sparser.setContentHandler(this);
            sparser.setErrorHandler(this);
            sparser.parse(file);
        }
        catch (Exception ex)
        {
            return false;
        }
        return true;
    }
```

```
    @Override
    public void startElement(String uri, String localName, String
qName, org.xml.sax.Attributes attributes) throws SAXException
    {
        System.out.print("<"+localName+">");
        if(localName.equals("id"))
            set=1;
        else if(localName.equals("service"))
            set=2;
        else if(localName.equals("destination"))
            set=3;
        else if(localName.equals("port"))
            set=4;
        else if(localName.equals("protocol"))
            set=5;
        else if(localName.equals("pass"))
            set=6;

    }

    @Override
    public void endElement(String uri,String localName,String
rawName) throws SAXException
    {
        System.out.println("</"+localName+">");
        set=0;
        if(localName.equals("service"))
            rl.setServices(sd);
    }

    @Override
    public void characters (char ch[], int start, int length)
throws SAXException
    {
        value=new StringBuffer("");
```

```
            value.append(ch,start,length);
            if(set==1)
            {
                System.out.println(value.toString());
                rl=new RuleDefinition(value.toString());
            }
            else if(set==2)
            {
                System.out.print(value.toString());
                sd=new ServiceDefinition();

            }
            else if(set==3)
            {
                System.out.print(value.toString());
                sd.setDestination(value.toString());

            }
            else if(set==4)
            {
                System.out.println(value.toString());
                sd.setPort(value.toString());
            }
            else if(set==5)
            {
                System.out.print(value.toString());
                sd.setProtocol(value.toString());
            }
            else if(set==6)
            {
                System.out.print(value.toString());
                rl.setPassword(value.toString());
            }
        }
    }
```

4.7.4 Cliente de Autentificación.

ANMAgent

Este agente es el encargado de enviar la información para dar acceso al usuario que se autentifica en la terminal.

Código ANMAgent.java

```
public class ANMAgent extends Agent
{
    public void behaviour()
    {
       System.out.println(this.getID()+" ready...");
    }

    public void newMessages()
    {
        Stack queue=getQueue();
        while(!queue.empty())
        {
            Wrapper msg=(Wrapper) queue.pop();
            System.out.println("from>>"+msg.getSender()
            +">>to>>"+msg.getReceiver()+">>"+msg.getObj());
        }
    }
}
```

ANMLogin.

Es la interfaz grafica que utiliza el usuario para autentificarse ante el sistemas, a través de este se comunica con el ANMAgent para que envié su autentificación.

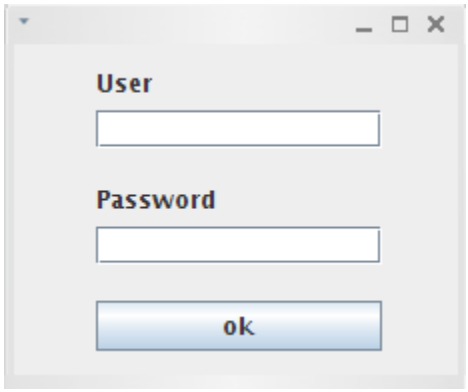

Fig. 4.4 Aplicación grafica para Autentificarse en el Sistema

Código ANMLogin.java

```
public class ANMLogin extends javax.swing.JFrame implements
AgentFrame
{
    ANMAgent agent;
    public ANMLogin() {
        initComponents();
    }

    private void initComponents() {
      ............
setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CL
OSE);
    }

    private void login(java.awt.event.MouseEvent evt)
    {
        try
        {
            agent.send("ANMaster", jTextField1.getText()+":"+new
```

```
String(jPasswordField1.getPassword())
+":"+InetAddress.getLocalHost().getHostAddress().toString());
        }
        catch(Exception ex){}
    }

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new ANMLogin().setVisible(true);
            }
        });
    }

    public void setAgent(Object agent)
    {
        this.agent=(ANMAgent) agent;
    }
}
```

4.7.5 Archivo de publicación de agentes para ANMSystem.

Por ultimo debemos solo debemos publicar los archivos de definicion de los agentes, por el lado del agente que controla el firewall el archivo ANMaster.xml que contiene lo siguiente:

```
<agent>
      <class>ANMS</class>
      <id>ANMaster</id>
</agent>
```

Del lado de los equipos clientes debemos publicar el archivo ANM-Client_ip_address.xml por cada equipo que vaya a utentificar con el firewall para acceder a los servidores.

```
<agent>
      <class>ANMAgent</class>
      <id>ANM-Client_ip_address</id>
      <gui>ANMLogin</gui>
</agent>
```

Como se muestra en el archivo de definicion este agente cuenta con interfaz grafica para comunicarnos con el agente dentro de nuestro contenedor, esta interfaz se accedara mediante el AplicacionLoader.

Capitulo V

Conclusiones y Trabajos Futuros.

5.1 Conclusiones

Como se presento al inicio de la investigación, en base a los problemas detectados, surgieron unas series de interrogantes sobre la factibilidad, las ventajas y desventajas del uso de agentes como Middleware para la construcción e integración de sistemas.

5.1.1 Requerimientos para la construcción e integración de sistemas mediante agentes.

Durante la etapa de análisis de las metodologías y herramientas para la construcción de sistemas distribuidos, una constante bien definida en cuestión de requerimientos fue el background técnico necesario para el desarrollo e integración de sistemas mediante el uso de agentes, ya que exige fuertes conocimientos en el área de redes, sistemas distribuidos y metodología de agentes como base, además del conocimiento en el área de dominio de la aplicación a desarrollar.

Por lo cual aunque el costo en infraestructura puede ser muy bajo, los requerimientos de tiempo y conocimiento pueden ser muy elevados, utilizando los esquemas y herramientas actuales.

5.1.2 Factibilidad de implementación.

Este punto es algo relativo ya que esto dependerá del área de dominio de la aplicación, así como si se trata de un desarrollo nuevo, o de una integración/migración de un sistema existente.

Tomando como base el sistema a desarrollar propuesto en esta investigación, y habiendo obtenido resultados favorables se puede determinar que el factor de éxito de un desarrollo de una nueva aplicación bajo esta tecnología dependerá del modelado del sistema, ya que requerirá una correcta identificación y abstracción de los elementos del mismo, así como de la asignación de responsabilidades a los agentes, un error durante las fases de análisis

y diseño puede ser catastrófica y mermar fuertemente la factibilidad del proyecto.

En el caso de una migración de tecnologías, si el sistema esta implementado bajo paradigma orientado a objetos con una fuerte cohesión en sus funcionalidades, la factibilidad incrementa ya que la metodología de agentes puede ser considera como una evolución de Orientado a objetos, aquí un factor de riesgo serian las limitantes del lenguaje y las características de los sistemas backend a los cuales se conecta el sistema. Si el sistema esta bajo paradigma estructurado, se puede considerar un desarrollo nuevo.

En el caso de la integración de sistemas existentes, los problemas son demasiados variados y son exponenciales en base al numero de aplicaciones que se deseen integrar, se pueden utilizar técnicas de wrapping o envolvimiento para facilitar la integración, pero los tiempos de análisis pueden ser muy grandes y complejos ya que requiere un estudio a fondo por cada aplicación buscando un punto de unión del cual obtener la información.

La factibilidad se ve reducida en gran medida para las migraciones o integraciones si no contamos con los códigos fuentes de alguna de las aplicaciones involucradas.

5.1.3 Tecnologías para implementación de sistemas multiagentes.

Las tecnologías presentadas dentro de esa investigación como RMI y JADE dan gran soporte para la implementación de aplicaciones distribuidas y con Agentes, con algunas ventajas y desventajas las cuales se presentan a continuación.

Ventajas

RMI es una extensión a los RPC (Remote procedure call) lo cual facilita la migración de aplicaciones que actualmente este funcionando bajo este tipo de esquema.

JADE es un framework de agentes FIPA complaint

Aprovechan las ventajas del lenguaje Java.

Los detalles de comunicación son transparentes para el programador

Permite el desarrollo rápido y fácil de objetos distribuidos (RMI) y sistemas multiagentes (jade)

Desventajas

No permite la fácil integración con sistemas heredados

Solo funcionan con el lenguaje java.

Aunque abstrae los detalles de comunicación al programador al ser diseñada para uso general, es compleja para usuarios noveles en el desarrollo de aplicaciones distribuidas y multiagentes.

Requiere de un Background técnico muy amplio en java, aplicaciones distribuidas y multiagentes para la implementación y mantenimiento de los sistemas desarrollados.

5.2 Framework desarrollado.

Se plantearon varios objetivos al inicio de la investigación uno de ellos era el análisis de las metodologías y herramientas para la construcción de sistemas distribuidos y multiagentes, y a través de los mismo se decidió construir un framework tomando las fortalezas de cada una de las metodologías y herramientas analizadas, en realidad se abarcaron mas herramientas para la construcción del mismo que no son tratadas dentro de este documento por cuestiones de tiempo y espacio, pero que vale la pena nombrar: JINI, JMS, JBOSS , WebSphere MQ y JADEX. El estudio de cada una de estos sistemas fue de vital importancia para tener un panorama claro de los puntos a atacar por el framework, además de ampliar los conocimientos en el área de sistemas distribuidos y multiagentes.

Como se describe a detalle en el capitulo III y IV de este documento, el framework fue

desarrollado con la idea principal de facilitar el desarrollo de aplicaciones con agentes, no solo desde un enfoque académico, si no también con un enfoque comercial; tratando de solucionar problemas que se presentan comúnmente dentro de las empresas, como el ejemplo planteado para el control de acceso a los servicios dentro de una red. Esta fue una de las razones por la cual el framework no cumple con todas las especificación FIPA para agentes, sino que es un híbrido en los esquemas planteados por la FIPA y de funcionalidades de algunas aplicaciones comerciales.

5.2.1 Ventajas.

Como se muestra a través de este documento la idea principal es proporcionar un conjunto de herramientas que faciliten el desarrollo de sistemas distribuidos con agentes, para esto se analizaron diversas herramientas libres como comerciales, a continuación se mencionaran algunos aspectos implementado en el framework y de donde se basaron las ideas.

El envió y recepción de mensaje implementa los esquemas uno a uno y uno a muchos, como se implementan dentro de los middlewares orientados a mensaje, con el objetivo de desarrollar sistemas con esquemas de comunicación asíncrona, la adopción de este tipo de esquema es fuertemente utilizado para la interconexión de sistemas heterogéneos, un ejemplo de aplicaciones de uso comercial que implementan este esquema son JMS [27] de SUN y WebSphere MQ de IBM [26].

Se definió que la comunicación dentro del sistema se realizara mediante paso de mensajes, el cual se basa en estructura definida por el ACL KQML [31], el cual define un formato de comunicación frecuente adoptado para el desarrollo de sistemas inteligentes. Que se defina este tipo de mensajes como estándar para la comunicación entre componentes, facilita el mantenimiento y desarrollo de funcionalidades del sistema.

El esquema de publicación de agentes, nos brinda la ventaja de poder agregar nuevos agentes al entorno de ejecución sin necesidad de detenerlo, la publicación se realiza a través de archivos XML el cual nos permite manipular a los agentes de una manera mas simple, esta función se basa en el esquema de publicación de componentes que brinda el servidor de aplicaciones JBOSS [28].

Aprovecha la ventajas que brinda el lenguaje java para cargar objetos de manera remota [22], esto nos permite que todas las aplicaciones descarguen de un punto central los objetos, facilitando las actualizaciones de estos objeto sin vernos en la necesidad de realizar tareas extras en los equipos clientes. Este esquema es muy utilizado en sistema de distribución de objetos como RMI [29] o JINI [24].

Se adopta XML como formato para los archivos de configuración del framework, esto facilita el entendimiento de los archivos al ser mas descriptivos, Esta idea se tomo de JADEX [25] el cual utiliza los archivos XML para definir los agentes del sistema.

Se brindan un conjunto de herramientas para facilitar el desarrollo de aplicaciones graficas, permitiendo desarrollar aplicaciones de uso común que utilicen agentes.

5.2.2 Desventajas.

La principal desventaja que tenemos con el framework es que es un desarrollo nuevo, que busca facilitar algunos problemas detectados en otras herramientas, aunque se basa en esquemas de trabajo probados no se apegan a todos los estándares de la industrias para este tipo herramientas, como JADE [26] que implementa las especificaciones dispuesta por la FIPA [30] para las plataforma multiagentes.

5.2.3 Contribución.

La principal contribución que se busca a través de la siguiente investigación es la formar

un marco referencial para la implementación de aplicaciones mediante el uso de agentes, mostrando las ventajas y desventajas asociadas a este tipo de sistemas, así como dar a conocer las herramientas actuales utilizadas para la implementación de estos tipos de esquemas.

Por otro lado se pretende sentar las bases para el desarrollo de una plataforma que sea una opción viable para el desarrollo de aplicaciones con agentes que su enfoque principal sea dar soluciones a problemas frecuentes dentro de la empresas. En el área académica el framework puede ser de gran utilidad para la demostración de ciertas metodologías para la construcción de sistemas distribuidos y multiagentes utilizada en el modelado del mismo.

Por otro lado el framework mismo es una base para el desarrollo de nuevas investigaciones, ya que su estructura interna permite la reutilización de sus diversos componentes en otros desarrollos que requieran, implementar esquemas de interconexión distribuida.

5.3 Trabajos Futuros.

El Framework aunque funcional, sigue en fase de prototipo ya que no se ha puesto a funcionar dentro de un ambiente de producción a gran escala, existen una serie de mejoras y áreas de oportunidad en la cual se puede desarrollar, el framework cumple con los objetivos planteados y presenta una gran panorama para su implementación tanto en el sector académico y comercial.

5.3.1 Mejoras al Framework.

Dentro de las tareas pendientes para adaptarlos a un ambiente de producción son:

1.-Mejorar el sistema de almacenamiento de mensajes. internamente el sistema tiene su propio esquema de almacenamiento de mensajes, el cual fue diseñado para

soportar de forma óptima la conexión de clientes simultáneos, pero no ha sido puesto a prueba bajo escenarios de alta demanda, se puede adaptar esta parte del sistema para utilizar JMS en caso de que el sistema actual no soporte múltiples clientes.

2.-Implementar un esquema de monitoreo de los contenedores registrados. Actualmente los contenedores que conforman la plataforma se dan de alta de manera dinámica y sin restricciones, por lo cual se requiere mejorar el sistema de control de los contenedores.

3.-Optimizar el sistema de publicación de agentes. En esta versión del framework se pueden cargar agentes en memoria leyendo los archivos de configuración XML pero no existe un esquema de eliminación.

4.-Implementar un esquema cifrado de comunicación. Para cuestiones de seguridad es importante implementar un esquema de comunicación cifrado entre los contenedores y el servidor de mensajes.

5.-Incluir Todas las Especificaciones FIPA. Actualmente el framework no cumple con todas las especificaciones marcadas por la FIPA, como el directorio de servicios.

6.-Implementar un sistema de publicación remota de agentes. Se puede publicar agentes en los contenedores pero se debe hacer de manera local, aprovechando que el sistema permite la carga de agentes de una ruta remota, seria de gran utilidad implementar un esquema de publicación remota.

5.3.2 Áreas de oportunidad

1.-Automatización de tareas del área de sistemas. Como se desmostro con la aplicación desarrollada con el framework, la automatización de tareas mediante el uso de agentes, es un campo de acción con muy grande beneficios, como por ejemplo se pueden implementar esquemas para automatizar los respaldos de los equipos y ejecución de script de manera remota, entre otras.

2.-Utilización de agentes en aplicaciones de automatización industrial, el sistema puede ser adaptado para equipos de capacidad limitada, aunque el sistema funciona bajo un esquema asíncrono puede ser optimizado para funcionar para aplicaciones en tiempo real.

3.-Adaptar el framework para dispositivos móviles. Esta parte es muy interesante por la tendencia hacia el uso de dispositivos móviles y a sus bajas características en hardware, se pueden implementar sistemas aprovechando las características del framework donde los cálculos y operaciones pesadas se realicen por agentes localizados en equipos de mayor capacidad.

[1] **Definición arquitectura cliente servidor.**
JAMES GILDARDO GUTIERREZ,2005.
Disponible en:
http://www.monografias.com/trabajos24/arquitectura-cliente-servidor/arquitectura-cliente-servidor.shtml

[2] **El modelo Cliente – Servidor.**
Disponible en:
http://www.unav.ws/common/fe/informes/diagnostico/cursos/sql/pagina_1.shtm

[3] **Cliente/Servidor y objetos: Guía de Supervivencia.**
Orfali, D.Harkey, 2002.
3ra.Ed McGraw -Hill.

[4] **La arquitectura cliente-servidor.**
Disponible en:
http://es.wikipedia.org/wiki/Cliente-servidor

[5] **Ventajas e inconvenientes de la arquitectura cliente-servidor.**
Disponible en:
http://www.csi.map.es/csi/silice/Global75.html

[6] **Framework for the organization of cooperative services in distributed client-server systems .**
Computer Communications, Volume 15, Issue 4, May 1992, Pages 261-269 .
Jürgen Nehmer, Friedemann Mattern.

[7] **Highly parallel distributed computing systems with optical interconnections Microprocessing and Microprogramming.**
Volume 27, Issues 1-5, August 1989, Pages 489-493

J.R. Just, R.S. Romaniuk

[8] **CONFLICT PATTERNS: TOWARD IDENTIFYING SUITABLE MIDDLEWARE.**
L. Davis R. Gamble,2009-06-03.
Disponible en:
www.sei.cmu.edu/library/assets/conflictpatterns.pdf

[9] **Middleware.**
Disponible en:
http://es.wikipedia.org/wiki/Middleware

[10] **Proceso distribuido, cliente/servidor y agrupaciones.**
David Luis la Red Martínez,2001.
Disponible en:
http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/MonogSO/PRODIS02.htm

[11] **RFC1057 - RPC: Remote Procedure Call Protocol specification.**
Sun Microsystems,Inc,1988.
Disponible en:
http://www.faqs.org/rfcs/rfc1057.html

[12] **Wrapping legacy systems for use in heterogeneous computing environments.**
Information and Software Technology, 43 (8), pp. 497-507.
Chia-Chu Chiang

[13] **Agentes móviles y corba.**
Ramón M. Gómez Labrador,1999
Disponible en:
http://www.informatica.us.es/~ramon/tesis/CORBA/Seminario-MASIF/

[14] **Invocación Remota de Metodos.**
Sun Microsystems,Inc.
Disponible en:
http://www.programacion.com/java/tutorial/rmi/2/

[15] **Servicio de búsquedas en una arquitectura de componentes GIS.**
Pedro Gustavo Picén Xancatl , 06-12-2000.
Disponible en:
http://catarina.udlap.mx/u_dl_a/tales/documentos/msp/picen_x_pg/

[16] **JAVA RMI.**
Revista Digital Universitaria,No. 1 ,Vol.2 ,31-03-2001
Rolando Menchaca Méndez, Félix García Carballeira.
Disponible en:
http://www.revista.unam.mx/vol.2/num1/art3/

[17] **RMI mano a mano con SSL: construyendo aplicaciones distribuidas seguras.**
Carlos Beltrán González,1999.
Disponible en:
http://www.programacion.com/bbdd/articulo/joa_rmissl/

[18] **Programación distribuida con RMI.**
Enrique Medina Montenegro,2003-09-05.
Disponible en:
http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=rmi

[19] **JADE framework.**
Ariel Monteserin,2007.
Disponible en:
http://www.exa.unicen.edu.ar/catedras/tmultiag/clase3.pdf

[20] **Curso de doctorado "Sistemas Multi-agentes".**
universidad politecnica de valencia.
Disponible en:
http://www.upv.es/sma/practicas/jade/Introducci%F3nJADE.pdf

[21] **JADE administrator's guide.**
Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB S.p.A., formerly CSELT),2007
Disponible en:
http://jade.tilab.com/doc/administratorsguide.pdf

[22] **JAVA Classloader.**
Sergio Talens Oliag,1999.
Disponible en:
http://www.uv.es/sto/cursos/seguridad.java/sjava.pdf

[23] **Java AWT: Delegation Event Model.**
Sun Microsystems,Inc,1997-02-03.
Disponible en:
http://java.sun.com/j2se/1.3/docs/guide/awt/designspec/events.html

[24] **JINI.**
Disponible en:
http://www.jini.org/wiki/Main_Page

[26] **JADE.**
Disponible en:
http://jade.tilab.com/

[25] **JADEX.**
Disponible en:
http://jadex.informatik.uni-hamburg.de/xwiki/bin/view/Main/

[26] **WebSphere MQ de IBM.**
Disponible en:
http://www-01.ibm.com/software/integration/wmq/

[27] **JMS.**
disponible en:
http://java.sun.com/products/jms/

[28] **JBOSS.**
Disponible en:
http://in.relation.to/Bloggers/IntroducingJBossAS600M1

[29] **RMI.**
Disponible en:
http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html

[30] **FIPA.**
Disponible en:
http://www.fipa.org/

[31] **Management of Multimedia Networks and Services .**
Raouf Boutaba, Abdelhakim Hafid,1998
1ra edi. Chapma & Hall

Apéndice

1. Codigo y material de referencia del Framework.

Por lo extenso de los codigo se ha incluido en formato digital esta informacion, puede encontrarlo en el disco anexo a esta tesina o puede descargalo de las siguiente URL rdominguez.eranet.com.mx/maestria/misc/framework.tar.gz o puede solictarlo via correo a las siguiente direccion rodrigo.dominguez@eranet.com.mx.

El contenido es el siguiente:

```
.
|-- Desarrollo
|   |-- ANMSystem
|   `-- Middleware
|-- Referecias
|   |-- Agentes
|   |-- Sistemas Distribuidos
|   `-- otros temas
|-- distribucion
|   |-- AContainer
|   |-- AMQServer
|   `-- ANMSystem
`-- tesina
```

La carpeta desarrollo contiene los proyectos en netbeans AMQServer, AgentContainer y AMQCommon; la carpeta referencia contiene una serie archivo que fueron utilizados durante esta investigacion; el directorio distribucion contiene los archivos para la instalacion del AMQServer, AgentContainer y ANMSystem, por ultimo el directorio tesina incluye una copia de este archivo en formato pdf.

2.Curriculum Vitae

Rodrigo Domínguez García

Paseos del Rosario # 14919 Col. Paseos de chihuahua, Chihuahua Chi.

Tel. 4819499 Cel. (614) 2535293.

E-Mail: rodrigo.dominguez@eranet.com.mx

Formación académica.

- Estudios en la facultad de ingeniería (UACH) dentro del plan de estudio maestría en ingeniería en sistemas computacionales (2006-2008), enfocándome al area de investigacion de diseño, desarrollo e integracion de sistemas distribuidos,basados en arquitectura cliente-servidor, orientada a servicios y sistemas multi-agentes. Herramientas utilizadas durante la formación son: J2SE, j2EE, RMI, JINI, Javaspaces, Jade, Jadex, Prolog, Lisp y Labview.

- Estudios realizados en la facultad de ingeniería (UACH) en le plan de estudios de ingeniería en sistemas computacionales opción software (2000-2005); mediante los cuales se adquirieron conocimientos sobre análisis, diseño, desarrollo e implantación de soluciones de software, así como la administración de sistemas operativos multiusuario e instalación y gestión de redes. Dentro de las diversas herramientas utilizadas durante el plan de estudios fueron las siguientes: paqueteria Microsoft Offices, lenguajes de programación turbo C++, Cobol, Delphi, manejador de BD mySQL, Oracle 9i, sistemas operativos windows, Linux , Unix y herramientas de configuracion de red basadas en equipo cisco.

Área laboral.

- **CIMAV** (15-01-2009 a la fecha) responsable de la división de computo científico. Responsabilidades del puesto: soporte en el desarrollo de aplicaciones para las áreas de investigación de física y química de materiales y de monitoreo ambiental, administración de los servidores de calculo científico, soporte técnico en sistemas Linux y herramientas de software libre.

- **ERANET** consultor independiente. cordinación de proyectos de miragración de servidores windows-linux, impartición de cursos de capacitación. areas de especialización en servicios Linux: Router, Firewall, Proxy(squid), Mailserver, webserver(Apache), Integracion con entornos Windows(samba), VPN, DHCP, LDAP, configuración de Primary Domain Controller. areas de especialización en desarrollo: administración de requerimientos con UML y Constucción de sistemas distribuidos con JAVA.

- **SYSCOM** (17-12-2004 a 07-01-2009) en el área de tecnologías de información como ingeniero de soporte a TI. Responsabilidades del puesto: administración del ERP empresarial, instalación y administración de los servidores empresariales, instalación de cableado estructurado y configuración de la red local, supervisión de los enlaces de comunicación, firewalls y vpns de la casa matriz y sucursales, instalación y soporte de enlaces VOIP vía radio, soporte tecnico a usuarios y supervisión de mantenimientos preventivos y correctivos a equipo de computo, diseño y desarrollo de componentes boundary para la expasion de funcionalidades del ERP, desarrollo de aplicaciones basadas en requerimientos de usuario, programacion de scripts para automatizacion de procesos dentro de lo servidores. Herramientas utilizadas durante la estancia: sistemas operativos windows 98, 2000 server ,XP, Unixware, Centos, Fedora y Ubuntu; utilerias

ThigthVNC, norton ghost, G4L(Ghost for linux), heartbeat, DRBD(Distributed Remote Block Device) , Microlite BackupEDGE; equipos switch capa 2 y 3 3com de la serie superstack 4200 y 5500, equipos fortinet de la serie fortigate 50,60A,300 y 300A.

- **SHCP** (16-02-2004 a 16-07-2004) brindando Servicio social dentro del área de tecnologías de la información. Con las funciones básicas de mantenimiento preventivo y correctivo de equipo de computo y a la administración del software institucional bajo la supervisión del L.S.C.A Jaime Estrada. Herramientas utilizadas durante la estancia: Sistema operativo windows 95, 98, 2000, ME y XP ; paqueteria Microsoft Office

CERTIFICACIONES

- IBM Linux administrator.
- ACE Certificate: Professional Developer of e-business Applications.
- IBM Certified Database Associate.

CURSOS Y APACITACIONES

- Taller de Redes. ITESM campus Chihuahua
- Taller de Redes inalambricas. ITESM campus Chihuahua
- Linux Basic and Installation Centro de Estandares Abiertos
- (IBM Training LX02).
- Linux System Administration I: Centro de Estandares Abiertos
- Implementation(IBM Training LX03).
- Linux Network Administration I:Centro de Estandares Abiertos
- TCP/IP services(IBM Training LX07).
- Linux Network Administration II:Centro de Estandares Abiertos
- Network Security and Firewalls(IBM Training LX24).
- Linux as a Web Server [Apache] Centro de Estandares Abiertos
- (IBM Training LX25).
- Linux Integration with Windows [Samba].Centro de Estandares Abiertos (IBM Training CY990).
- Linux Systems Administration project. Centro de Estandares Abiertos
- Object Oriented Programming with C++. Centro de Estandares Abiertos (IBM Training CY410).
- Core Java. Centro de Estandares Abiertos (IBM Training CY420).
- DB2 UDB programming and Store Procedure Centro de Estandares Abiertos (IBM Training CY430).
- Software Engineering. Centro de Estandares Abiertos (IBM Training CY440).
- OOAD using UML. Centro de Estandares Abiertos (IBM Training CY450).

- Web Programming I. Centro de Estandares Abiertos
- (IBM Training cy710).
- Web Programming II. Centro de Estandares Abiertos (IBM Training cy720).
- E-Business Application Developer and web project. Centro de Estandares Abiertos (IBM Training cy460).
- E-Business Technology Fundamentals. Centro de Estandares Abiertos (IBM Training cy750).
- E-Commerce Fundamentals. Centro de Estandares Abiertos (IBM Training cy820).
- Enterprise Application Development Using XML. Centro de Estandares Abiertos (IBM Training cy730).
- Introduction to WSAD and Enterprise Computing. Centro de Estandares Abiertos (IBM Training cy760).
- Enterprise JavaBeans(EJBs) II. Centro de Estandares Abiertos (IBM Training cy780).
- E-business Security Overview. Centro de Estandares Abiertos (IBM Training cy740).
- E-Business Application Development Project. Centro de Estandares Abiertos (IBM Training cy770).
- Essentials of Rational Unified Process V7.0. Centro de Estandares Abiertos (IBM Training RP401).
- Fundamentos de CMMI y Rational Unified Process.Centro de Estandares Abiertos (IBM Training CMMI VD RUP).
- Essentials of Visual Modeling with UML 2.0. Centro de Estandares Abiertos (IBM Training RD201).
- Mastering Object-Oriented Analysis and Design. Centro de Estandares Abiertos with UML 2.0 (IBM Training RD601).
- Mastering Requirements Management with use cases. Centro de Estandares Abiertos (IBM Training RR611).
- Rational Principles of Software Testing for Testers. Centro de Estandares Abiertos (IBM Training RT101).
- Essentials of Rational ClearCase for Windows. Centro de Estandares Abiertos (IBM Training RS501).

IDIOMAS

- Ingles: Hablado 80% ,Escrito 60%, Leido 80 %