Benemérita Universidad Autónoma de Puebla

Facultad de Ciencias de la Computación

Infraestructura genérica para agentes móviles para aplicaciones distribuidas bajo ProActive

Tesis

Que para obtener el grado de:

Maestro en ciencias de la Computación

Presenta:
Margarita Márquez Tirso

Asesores

Dr. Denis Caromel

M.C. David Eduardo Pinto Avendaño

Puebla, Pue., Mayo de 2004

Agradecimientos a:

Dios, que aun en mis peores travesuras está conmigo

Mis amados padres por que a pesar de todo lo son

Mis más grandes logros de mi vida: mi marido Víctor Javier por su infinito amor
Y
Mi hija Joyce por su paciencia y amor

Mi Suegra Lidia por su inapreciable ayuda

Mis hermanos Angela, Rafael, Gabriel, Pilar, Francisco por que de ellos siempre aprendo

Mis asesores Denis Caromel y David E.Pinto por su voto de fe

Mis sinodales Darnes Vilariño, Fabiola López por sus valiosas aportaciones

Mis amigos:
Adriana Hernández por el crecimiento que ambas tuvimos en este nuestro concepto de amistad

Jesús Velásquez, MariaElena Pineda, HildaAlicia Romero, Rocío Villamil, Adriana Sandoval por creer siempre en mi.

Marcos y Beatriz por que siempre están cuando deben estar

Gustavo y Yadira por que sin su pesimismo ya hubiera perdido la fe

Jose J.Lavalle, Jesús García, Miguel Ángel León por que siempre fueron asertivos conmigo

Sofía Paniagua por el apoyo moral siempre presente

David Pinto Júnior por iluminar con su presencia

Javier Caldera, por su amistad

Yenssen y Angeles por su sentido del humor

Miguel Ayar por que siempre tiene una sonrisa

Héctor López por que me enseñó la tabla de clasificación de los amigos

Víctor H. López, por que me ayudó al principio de este reto.

Claudia por que me enseñó que la diplomacia no esta peleada con la educación
Doña Josefina, por que nunca ha dudado en dar la mano

A la MC. Yolanda por sus deseos de buena suerte
Alejandra por su amistad sincera
Dr. Carlos Celaya por esa fe ciega

Y si alguien me falto, no dude que anda perdido en mis pensamientos, pero eso no implica que no le agradezca el tiempo compartido.

Contenido

Figuras

PREFACIO

Con la utilización generalizada de computadoras personales, estaciones de trabajo y redes de área local, los sistemas distribuidos cada vez son más comunes. Este tipo de sistemas es inherentemente más complejo que los no distribuidos y la programación de aplicaciones que se aprovechan de esta distribución sufre también de mayor complejidad. Actualmente se han propuesto distintos sistemas operativos y lenguajes de programación para reducir la complejidad de éstos.

Las aplicaciones distribuidas necesitan de manera habitual compartir datos entre entidades remotas y el hecho de compartir se consigue accediendo a datos remotos compartidos o copiándolos entre las entidades participantes.

Los sistemas y lenguajes antes de la aparición de los sistemas distribuidos prohibían en general el compartir datos de forma distribuida; para poder lograr el efecto de distribución, estos sistemas proporcionaban dos niveles de soporte, uno para los datos contenidos totalmente en una máquina y otro para los datos que eran compartidos, superando así las barreras entre diferentes máquinas.

Actualmente hay sistemas que proporcionan una forma limitada de compartir datos, esto es logrado mediante la replica de pequeñas cantidades de datos; un ejemplo de esto es incluir los mismos en mensajes o a través de parámetros en llamados a procedimientos remotos. Este tipo de replicación se denomina "replicación de grano fino".

Por otro lado, el paradigma de orientación a objetos (OO) se presenta como un paradigma estandarizado el cual es utilizado ampliamente en el desarrollo de aplicaciones por las ventajas que éste presenta; una de éstas es el uso de un único paradigma en las diferentes fases del ciclo de vida del software [BOO01], los sistemas resultantes son más fáciles de entender y de desarrollar, puesto que reducen los saltos semánticos al aplicar los mismos conceptos en todas las fases.

Aunado a la tecnología OO surge el término de agente el cual es una evolución de los clásicos objetos; particularmente los agentes móviles presentan un nuevo enfoque para el desarrollo de la arquitectura e implementación de sistemas distribuidos.

Entre los actuales retos en los sistemas distribuidos se encuentra el obstáculo que representa el desarrollo de aplicaciones distribuidas multiplataforma; para poder lograr esto hoy día se hace uso de la máquina virtual de Java y de una "*Aplication Program Interface*" (API), lo que permiten aislar programas Java del hardware y además aprovechar muchas otras ventajas que hacen particularmente apropiado el escenario para la tecnología de agentes móviles.

El comportamiento distribuido de Java permite un mejor balance de carga de trabajo en el sistema. A través de los Applets y de la utilización de RMI (Remote Method Invocación) se accede de forma simple y directa al manejo de objetos Java distribuidos. Java también provee múltiples hilos concurrentes que llevan a cabo tareas distintas en un programa. Sin

embargo existe aún un gran vacío entre las aplicaciones distribuidas y multihilos de Java, las cuales prohíben la reutilización de código para construir aplicaciones distribuidas a partir de aplicaciones multihilos.

Tanto JavaRMI como JavaIDL (API heterogénea para llamadas a procedimientos remotos) son ejemplos de bibliotecas de objetos distribuidos en Java, que dificultan el trabajo del programador ya que éste necesita realizar modificaciones profundas al código existente para convertir los objetos locales en objetos que puedan ser accesibles de manera remota.

En estos sistemas, los objetos remotos son accesibles a través de una interfase mas especializada; por otro lado estas bibliotecas de objetos distribuidos no permiten el polimorfismo entre los objetos locales y los remotos, considerando que el polimorfismo es un requerimiento primario para un ambiente de trabajo de metacómputo. El polimorfismo es necesario por que permite al programador concentrarse en el modelo y en las características algorítmicas más que en las tareas de bajo nivel tales como la distribución de objetos, mapeo y balance de carga.

Debido a lo anterior ProActive PDC entra en escena. ProActive es una biblioteca para cómputo distribuido, concurrente y paralelo OO en Java, en constante crecimiento y la carencia de las primitivas mínimas necesarias para la administración de agentes móviles, hacen que el objetivo de ésta tesis sea ampliar el contexto de la biblioteca en el área de agentes móviles.

A continuación se presentan los objetivos y la distribución de los capítulos de la presente tesis.

En esta tesis se tiene como objetivo:

- Desarrollar una Infraestructura genérica de agentes para aplicaciones distribuidas bajo ProActive, en la que se integren un conjunto de primitivas para el uso de agentes móviles en aplicaciones distribuidas, mostrando los beneficios de los sistemas distribuidos, la tecnología OO y el nuevo paradigma de Agentes.

Y como consideración:

- Promover el estudio de la biblioteca para futuras aportaciones a la misma.

Esta tesis se desarrolla como una colaboración de la Facultad de Ciencias de la Computación de la Benemérita Universidad de Puebla dentro del proyecto de consorcio ObjectWeb, el cual es una comunidad de Software Open Source Middleware[1] creada a finales de 1999 por FRANCE TELECOM R&D, Bull y por el INRIA (Instituto Nacional de Investigación en Informática y Automatización).

El contenido de esta tesis se agrupa en cuatro secciones: antecedentes, solución, prototipo y conclusiones. Cada una de estas se constituye en una fase de trabajo encaminada a la

[1] Software que permite el acceso a los recursos de una red

solución del objetivo de esta tesis, desde la presentación hasta la construcción de un prototipo.
La sección de antecedentes está distribuida a través de los capítulos 1, 2 y 3.

El capitulo 1, presenta la evolución y la situación actual de la construcción de sistemas distribuidos orientados a objetos., analizando las diferentes herramientas para el computo distribuido con respecto a ProActive.

En el capítulo 2, se describen los fundamentos de la biblioteca ProActive PDC, con el fin de introducir al lector con las bondades de dicha biblioteca.

El paradigma de los agentes es presentado en el capítulo 3, los conceptos abarcan desde la definición de agentes móviles hasta las diferentes infraestructuras de agentes propuestas y sus aplicaciones.

La sección dedicada a la solución de esta tesis se aborda en el capítulo 4. En este capítulo se modela la infraestructura genérica para agentes móviles haciendo uso del lenguaje modelado unificado (UML), a través de diagramas de casos de uso, de clases, de secuencias, de actividades, de secuencias y de despliegue. Logrando con esto una documentación actual y consistente a través de una notación estándar.

Y en el capítulo 5 se presenta el prototipo de la infraestructura genérica para agentes móviles desarrollada.

En la parte final de este documento se presentan las conclusiones del trabajo y las líneas de investigación que se abren a partir del punto en que finaliza la tesis.

Capítulo 1 Herramientas para el desarrollo de sistemas distribuidos

1.1 Introducción

Actualmente las herramientas para el desarrollo de sistemas informáticos presentan un avance significativo, por la parte del software se están atacando temas que retan la capacidad del procesamiento de computadoras independientes y que están dispersas geográficamente, estos temas son los que se abordan en el desarrollo de sistemas distribuidos. En la construcción de estos sistemas, el alcance es la capacidad de procesamiento para resolver un problema concreto y se está hablando entonces particularmente de **aplicaciones distribuidas**.

Un **sistema distribuido** está formado por un conjunto de computadoras autónomas unidas por una red de comunicaciones y software de sistemas distribuidos. El software de sistemas distribuidos permite a las computadoras coordinar sus actividades y compartir los recursos del sistema como son el hardware, el software y los datos. Los usuarios de un sistema distribuido bien diseñado deberían percibir una única imagen de computación, aun cuando dicha imagen podría estar formada por un conjunto de computadoras localizadas de manera dispersa (ver Figura 1.1).

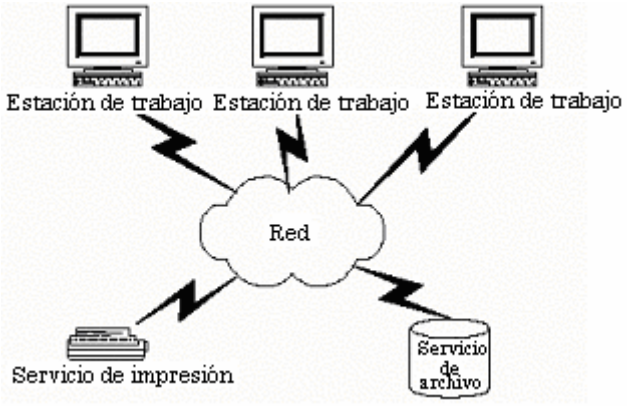

FIGURA 1.1 ESTRUCTURA DE UN SISTEMA DISTRIBUIDO

Uno de los problemas fundamentales en los sistemas distribuidos es el software. Actualmente no existe mucha experiencia en el diseño, implantación y uso de software distribuido.

Todo lo anterior ha dado pie al desarrollo de diferentes "*middlewares*" para el desarrollo de aplicaciones distribuidas. En este capitulo se hace un recorrido por estos sistemas resaltando sus funcionalidades así como sus inconvenientes, los "*middlewares*" son representativos de las diferentes tendencias actuales.

1.2 DCE y DC++

DCE *(Distributed Computing Environmen*t) de *Open Software Foundation* [OSF92], ahora denominada *Open Grou*p, es un conjunto integrado de herramientas y servicios que soportan el desarrollo de aplicaciones distribuidas, de entre los que se destacan los **llamados a procedimientos remotos** (RPC, *Remote Procedure Call*), el **servicio de directorio de celda** (CDS, *Cell Directory Servic*e), los **servicios de directorio globales** (GDS, *Global Directory Service*s), el **servicio de seguridad**, los **hilos** DCE, y el **servicio de archivos distribuidos** (DFS, *Distributed File Servic*e).

DCE es una tecnología de tipo *"middlewar*e" que no tiene sentido de existir por sí misma, sino más bien como un extra de un sistema operativo. Funciona en diferentes tipos de computadoras, sistemas operativos y redes, facilitando la portabilidad del software al ocultar las particularidades del entorno en que se ejecuta.

1.2.1 Modelo de programación

El modelo de programación de DCE es el **modelo cliente/servido**r. Las dos facilidades que ofrece DCE y que no pueden considerarse como servicios son los llamados a procedimientos remotos (RPC, *Remote Procedure Call*) y los hilos.

1.2.2 RPC

El **llamado a procedimientos remotos** es la facilidad que hace posible a un programa cliente acceder a un servicio remoto invocando simplemente un procedimiento local. Es responsabilidad del sistema de RPC ocultar todos los detalles a los clientes y servidores: localizar el servidor correcto, construir y transportar los mensajes en ambas direcciones y realizar todas las conversiones de tipos necesarias entre el cliente y servidor, evitando las posibles diferencias de las arquitecturas en las que se ejecutan ambos.

1.2.3 Hilos

Es común que los sistemas distribuidos utilicen RPC e hilos.
Al iniciar un hilo servidor, "S", éste exporta su interfaz al informarle de ésta al núcleo; la interfaz define los procedimientos que puede llamar, sus parámetros, etc.
Al iniciar un hilo cliente, "C", éste importa la interfaz del núcleo:

- Se le proporciona un identificador especial para utilizarlo en la llamada.
- El núcleo sabe que "C" llamará posteriormente a "S":
    - Crea estructuras de datos especiales para prepararse para la llamada.

Una de las estructuras es una pila de argumentos compartida por "C" y "S", que se asocia de manera lectura / escritura en ambos espacios de direcciones.

Para llamar al servidor, "C":

- Coloca sus argumentos en la pila compartida mediante el procedimiento normal de transferencia.
- Hace un señalamiento al núcleo colocando un identificador especial en un registro.

El núcleo:

- Detecta esto y deduce que es una llamada local.
- Modifica el mapa de memoria del cliente para colocar éste en el espacio de direcciones del servidor.
- Inicia el hilo cliente, al ejecutar el procedimiento del servidor.

La llamada se efectúa de tal forma que:

- Los argumentos se encuentran ya en su lugar:
    - No es necesario su copiado u ordenamiento.
    - La RPC local se puede realizar más rápido de esta manera.

Los hilos permiten a los RPC salientes obtener un mejor rendimiento, logrando que un cliente pueda acceder varios servidores al mismo tiempo en lugar de un servidor a la vez. Esta característica es interesante en servidores especializados en donde los clientes pueden dividir un RPC complejo en varios llamados RPC concurrentes más simples, ver Figura 1.2.

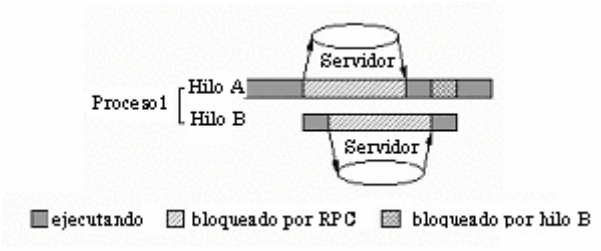

FIGURA 1.2 RPC UTILIZANDO HILOS

1.2.4 DC++

DC++ [SMM93] es un **entorno distribuido orientado a objetos** construido sobre DCE. En oposición a la visión procedimental de DCE (llamada a procedimiento remoto), soporta un modelo de objetos uniforme, invocación de objetos de grano fino independiente de la localización, paso de parámetros usando referencias remotas, migración dinámica de objetos e integración con el lenguaje C++. Los servicios fundamentales de DCE que utiliza DC++ son los hilos, las RPC.

Los **objetos distribuidos** pueden ubicarse en diferentes nodos del sistema distribuido y manipulan referencias locales y remotas.

Las referencias remotas se implementan con una dirección proporcionada por un **representante** *(proxy)*. El representante contiene una pista de la localización del objeto referenciado y le reenvía las invocaciones de manera transparente utilizando RPC. Se instalará un representante en todo nodo que conozca la existencia de un objeto remoto.

Este ocurre cuando se pasa una referencia a un objeto remoto como parámetro de una invocación. También ocurre cuando un objeto migra y tiene referencias a objetos remotos: en el nodo destino se tendrán que instalar representantes para todas las referencias.
La migración de objetos se solicita invocando un método generado automáticamente para todos ellos. Una vez movido el objeto, en el nodo original se dejará un representante del mismo. Como prerrequisito para la migración de un objeto, se asume que la clase a la que pertenece el objeto está disponible en el nodo destino basándose en un servicio de replicación de clases.

1.2.5 Inconvenientes

- El uso de un paradigma orientado a procedimientos, excluye un funcionamiento orientado a objetos. Aunque proporcione un entorno orientado a objetos, la construcción de DC++ está basada en el carácter procedimental de DCE.
- Los servicios que proporcionan los servidores en DCE y los objetos distribuidos en DC++, se tienen que describir utilizando un lenguaje de definición de interfaces IDL.
- La construcción de aplicaciones, en el caso de DC++, está restringida al uso del lenguaje C++.
- Cualquiera de las dos variantes (DCE y DC++) no dejan de ser una capa de software o "*middleware"* que se ejecuta sobre un sistema operativo tradicional.

1.3 CORBA

CORBA *(Common Object Request Broker Architectur*e) [OMG99] es una especificación definida por el OMG *(Object Management Grou*p) para la creación y uso de objetos remotos, cuyo objetivo en proporcionar interoperabilidad entre aplicaciones en un entorno distribuido y heterogéneo.

Es conocido como un tipo de *"middlewar*e", ya que no realiza las funciones de bajo nivel, a pesar de que debe funcionar sobre sistemas operativos tradicionales, realiza muchas de las operaciones que tradicionalmente se han considerado del dominio de los sistemas operativos para entornos distribuidos.

1.3.1 Objetos CORBA

Los **objetos CORBA** se diferencian de los objetos de los lenguajes habituales de programación debido a que pueden estar localizados en cualquier lugar de la red, pueden ejecutarse en cualquier plataforma y pueden estar escritos en cualquier lenguaje.

Un cliente puede utilizar un objeto CORBA sin saber donde está ni en qué lenguaje ha sido implementado (ver Figura 1.3).

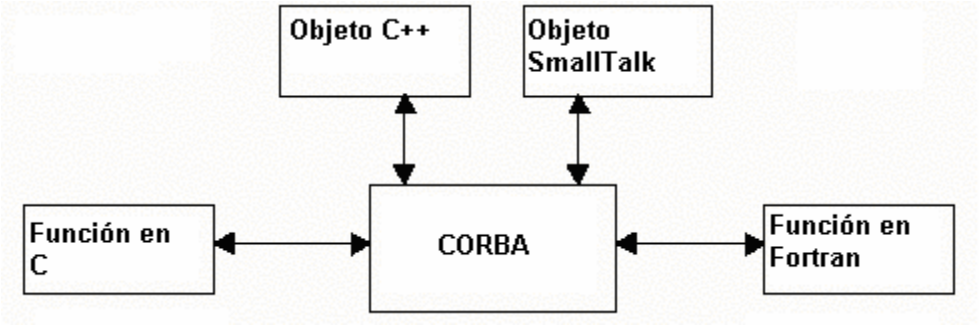

FIGURA 1.3 LA FUNCIÓN DE CORBA.

1.3.2 El ORB

El corazón de CORBA es su ORB *(Object Request Broke*r). El ORB es el responsable de buscar la implementación del objeto servidor, prepararlo para recibir la petición, poner en contacto el cliente con el servidor, transportar los datos (parámetros y valores de retorno) entre uno y otro transformándolos adecuadamente.

En definitiva es el responsable de que ni el cliente ni el servidor necesiten conocer la localización ó el lenguaje de implementación del otro ver Figura 1.4.

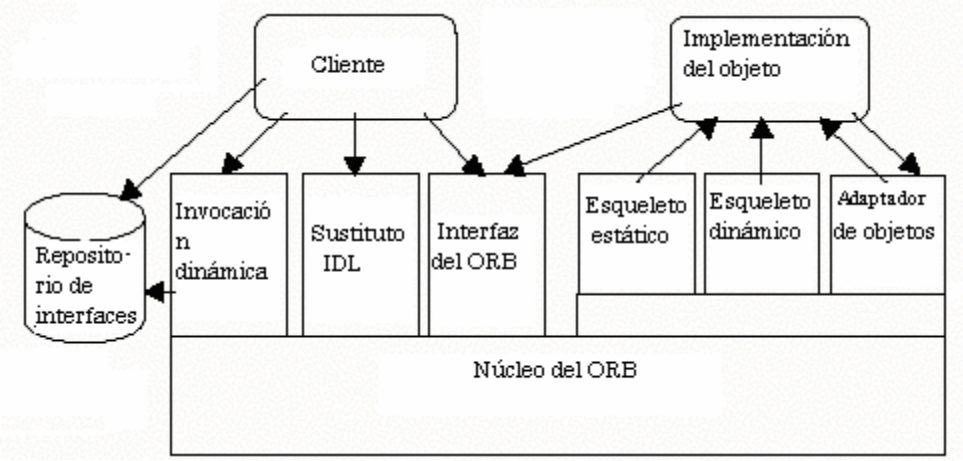

FIGURA 1.4 ESTRUCTURA DEL ORB DE CORBA.

1.3.3 Definición de objetos CORBA

Los objetos CORBA se tienen que definir con el **lenguaje de definición de interfaces IDL** *(Interface Definition Language*) que, como su propio nombre indica, se limita a definir la interfaz del objeto y no su implementación. La **definición del objeto** incluye la definición de sus atributos, sus métodos (denominados operaciones) y las excepciones que eleva. La **implementación del objeto** se tiene que realizar con algún lenguaje de programación que tenga enlaces con IDL (en la actualidad existen enlaces con lenguajes como C, C++, Java, Smalltalk, Ada, etc.).

IDL proporciona herencia múltiple de interfaces, de manera que las interfaces derivadas heredan las operaciones y los tipos definidos en las interfaces base. Todas las interfaces derivan de una interfaz raíz, denominada "*Object*", la cual proporciona servicios que son

comunes a todos los objetos CORBA, como duplicación y liberación de referencias a objetos, etc.

La compilación de una definición de objetos en IDL genera, entre otras cosas, un sustituto *(stu*b) y un esqueleto *(skeleto*n), para el cliente y servidor, respectivamente. El sustituto crea una petición de servicio al ORB a instancias del cliente y representa al objeto servidor. El esqueleto, por su parte, entrega las peticiones a la implementación del objeto CORBA. Es posible utilizar para la definición de los objetos una serie de tipos básicos y construidos que no tienen la categoría de objetos y que son similares a otros tipos encontrados en la mayoría de los lenguajes de programación: *char, boolean, array, struct*, etc.

1.3.4 El repositorio de interfaces

Las interfaces de objetos se almacenan en un **repositorio de interfaces** (IR, *Interface Repository*), que proporciona un almacenamiento persistente de las declaraciones de interfaces realizadas en IDL. Los servicios proporcionados por un IR permiten la navegación por la jerarquía de herencia de un objeto y proporcionan la descripción de todas las operaciones soportadas por un objeto.

La función principal del IR es proporcionar la información de tipos necesaria para realizar peticiones utilizando la **Interfaz de Invocación Dinámica**, aunque puede tener otros propósitos, como servir de almacenamiento de componentes reutilizables para los desarrolladores de aplicaciones.

1.3.5 Referencias

Para que un cliente pueda realizar una petición a un objeto servidor, deberá utilizar una **referencia** al objeto. Una referencia siempre refiere el mismo objeto para la que fue creada, durante tanto tiempo como exista el objeto. Las referencias son tanto inmutables como opacas, de manera que un cliente no puede "entrar" en una referencia y modificarla. Sólo el ORB sabe que es lo que hay "dentro" de la referencia.

Cuando se pasan objetos como parámetros en invocaciones a métodos, lo que realmente se pasan son referencias a dichos objetos. El paso de objetos como parámetro es, por tanto, por referencia.

1.3.6 Paso de parámetros por valor

En la revisión de CORBA (revisión 2.3, de Junio de 1999) se introdujo la propuesta de **objetos-por-valor** *(objects-by-valu*e), que extienden el modelo de objetos de CORBA tradicional para permitir el paso de objetos por valor dentro de los parámetros de los métodos. Para conseguirlo se introduce un nuevo tipo IDL denominado el tipo **valor** *(valu*e).

Cuando un ORB encuentra un objeto de tipo "valor" dentro de un parámetro, automáticamente pasa una copia del estado del objeto al receptor. En contraste, el ORB pasará por referencia cualquier objeto declarado vía una interfaz IDL.

Un tipo "valor" se puede entender entre una interfaz IDL y una estructura (del estilo de las del lenguaje C). La sintaxis del valor permite especificar algunos detalles de implementación (por ejemplo, su estado), que no forman parte de una interfaz IDL. Además, se pueden especificar métodos locales para operar con dicho estado. A diferencia de las interfaces, los métodos de los tipos "valor" no pueden ser invocados remotamente.

1.3.6 Inconvenientes

- El modelo de objetos proporcionado por CORBA y un nuevo tipo de objeto con paso de parámetros por valor, perjudica la uniformidad en la orientación a objetos.
- En el código del usuario se mezclan partes relacionadas con el dominio del problema resolver y otras relativas a la gestión de la comunicación por lo que la programación no es transparente.
- A pesar de su gran éxito para la construcción de sistemas distribuidos, CORBA no deja de ser una capa añadida al sistema operativo.
- El hecho de que los objetos puedan estar programados en diferentes lenguajes de programación y para diferentes arquitecturas y sistemas operativos imposibilita, prácticamente, su migración. Únicamente la utilización de CORBA con un lenguaje de programación independiente de la plataforma, como Java, permite cierto grado de movilidad a los objetos.

1.4 DCOM

COM *(Component Object Model*) [MIC95] es la tecnología de definición y manipulación de componentes de Microsoft que proporciona un modelo de programación y un estándar binario para los mismos. DCOM [MIC98] (Distributed COM) es la tecnología que extiende de COM para permitir a los objetos componentes residir en máquinas remotas y está disponible desde la aparición de Windows NT 4.0. A partir de ahora se utilizarán los términos COM y DCOM indistintamente.

1.4.1 Modelo de objetos

Un **objeto COM** se define en términos de las interfaces individuales que soporta (una o más) y que están definidas en la clase (objeto clase) a la que pertenece. Cada interfaz está identificada por un **identificador único** (IID, *Interface Identifier*), que es un caso particular de **identificador global y único** (GUID, *Global Unique Identifier*).

Los GUID son valores de 128 bits que se garantizan únicos estadísticamente en el espacio y en el tiempo.

Las interfaces son el único medio de interactuar con un objeto COM. Un cliente que desea utilizar los servicios de un objeto habrá de obtener primero un puntero a una de sus interfaces.

Los **objetos clase** implementan una o más interfaces y se identifican por un **identificador de clase** (CLSID, *Class Identifie*r). Los CLSID son también un caso particular de GUID. Con el fin de que un cliente pueda crear un objeto COM, es necesario describir su clase

utilizando el **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). La compilación de dicha descripción genera un representante *(prox*y) para los clientes y un sustituto *(stub*) para el servidor.

COM no proporciona la herencia como instrumento para lograr la reutilización. En su lugar proporciona los **mecanismos de contención y agregación**. El polimorfismo se consigue cuando diferentes clases soportan la misma interfaz, permitiendo a una aplicación utilizar el mismo código para comunicarse con cualquiera de ellas.

1.4.2 Interoperabilidad entre objetos COM

El objetivo principal de COM es proporcionar un medio por el que los clientes pueden hacer uso de los objetos servidores, sin tener en cuenta que pueden haber sido desarrollados por diferentes compañías utilizando diferentes lenguajes de programación.

Con el fin de lograr este nivel de interoperabilidad, COM define un **estándar binario**, que especifica cómo se dispone un objeto en memoria principal en tiempo de ejecución. Cualquier lenguaje que pueda reproducir dicha disposición en memoria podrá crear objetos COM (ver Figura 1.5).

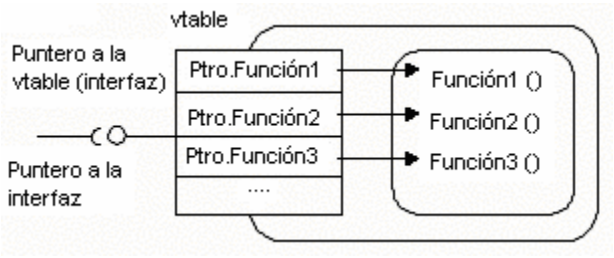

FIGURA 1.5 DISPOSICIÓN DE UN OBJETO COM EN MEMORIA.

Además del objetivo de la interoperabilidad, COM tiene los siguientes objetivos:

- Proporcionar una solución para los problemas de versiones y evolución.
- Proporcionar una visión del sistema de los objetos.
- Proporcionar un modelo de programación singular.
- Proporcionar soporte para capacidades de distribución.

1.4.3 Modelo de programación

En el modelo de programación COM, los clientes COM se conectan a uno o más objetos COM. Cada objeto COM expone sus servicios a través de una o más interfaces, que no son más que agrupaciones de funciones relacionadas semánticamente.

La implementación compilada de cada objeto COM está contenida en un módulo binario (exe o dll) denominado **servidor CO**M. Un único servidor COM es capaz de contener la implementación compilada de varios objetos COM.

Un servidor COM puede estar enlazado al proceso cliente *(in-process server*), puede ejecutarse en un proceso distinto del cliente pero en la misma máquina *(local server*) o

bien, puede ejecutarse en un proceso separado en una máquina distinta, incluso en un sistema operativo distinto *(remote server*). Para la comunicación con objetos situados en espacios de direcciones distintos del espacio de direcciones del cliente, se utilizan intermediarios en la forma de representantes y sustitutos.

El modelo de programación COM define que un servidor COM debe exponer objetos COM, un objeto COM debe exponer sus servicios y un cliente COM debe usar los servicios de objetos COM.

Para comunicarse con un objeto que no es local, COM emplea un mecanismo de comunicación entre procesos que es transparente a la aplicación, incluso en lo relativo a la localización del objeto.

1.4.4 Ciclo de vida

Todos los objetos COM tienen que implementar una interfaz particular, denominada "**IUnknown"**. Esta interfaz es el corazón de COM y es utilizada para negociación de interfaces en tiempo de ejecución (preguntar al objeto qué interfaces soporta y obtener punteros a las mismas), gestión del ciclo de vida del objeto y agregación.

Cada objeto mantiene una **cuenta de referencia** que es incrementada cada vez que un cliente solicita un puntero para una interfaz o pasa una referencia al mismo. Cuando la interfaz deja de ser utilizada por el cliente, la cuenta de referencia se decrementa. Las operaciones de incremento y decremento son realizadas de manera explícita por el cliente invocando funciones pertenecientes a la interfaz "*IUnknown*".

1.4.5 COM+

COM+ [KIR97] se presentó como una evolución de COM y como base del sistema operativo de Microsoft, Windows 2000.

COM+ viene a ocultar la mayor parte de las tareas que en COM resultaban tediosas y dificultosas para los programadores, como el control del ciclo de vida, la negociación de interfaces, etc. A pesar de todo, el modelo básico de objetos sigue siendo el de COM.

El conjunto de servicios que introduce COM+ está orientado a la construcción de aplicaciones empresariales, de tal forma que los programadores se concentran en la escritura de la lógica de negocio y no tienen que perder tiempo escribiendo infraestructura u otros servicios.

Las características principales que se pueden encontrar en COM+ son: servidores, transacciones, seguridad, administración, equilibrado de carga, componentes encolados *(queued components*) y eventos.

1.4.6 Inconvenientes

- COM proporciona un concepto de objeto muy vago y alejado. La herencia no está contemplada en el modelo de objetos. Los objetos no tienen una identidad propiamente dicha.

- Los objetos no son entidades autónomas, sino que necesitan de los procesos, que les proporcionan un entorno en el que ejecutarse.
- Programación compleja. Los servidores deben realizar tareas adicionales a las propias del dominio del problema, con el fin de que su objeto pueda ser utilizado por los clientes. Los clientes, por su parte, sólo pueden utilizar punteros para referenciar los objetos a través de alguna de sus interfaces.

1.5 RMI de Java

RMI *(Remote Method Invocation*) [SUN98] fue diseñado para permitir la invocación de métodos remotos de objetos entre distintas máquinas Java de manera transparente.
Integra directamente un modelo de objetos distribuidos en el lenguaje Java a través de un conjunto de clases e interfaces.

1.5.1 Objetos RMI

Un **objeto RMI** es un objeto cuyos métodos pueden ser invocados desde otra máquina Java, incluso a través de una red. Cada objeto remoto implementa una o más **interfaces remotas** que especifican qué operaciones pueden ser invocadas por los clientes. Cualquier otra operación pública que tenga el objeto pero que no aparezca en la interfaz no podrá ser utilizada por los clientes remotos. Los clientes invocan dichos métodos exactamente igual que si fueran métodos locales, quedando ocultos los detalles de la comunicación. Se utilizan los denominados **sustitutos** *(stub*) y **esqueletos** *(skeleton*), que actúan de intermediarios entre los objetos local y remoto. Los sustitutos y los esqueletos son generados de manera automática por el compilador *rmic*.

1.5.2 Modelo de programación

Desde el punto de vista del programador, los objetos remotos se manipulan de la misma manera que los locales, a través de referencias. Realmente, una **referencia** a un objeto remoto apunta a una referencia a un sustituto local que lo representa, y que es el encargado de transmitir los parámetros de la invocación a la computadora remota y recibir de él el valor de retorno. La manipulación de objetos remotos debe tener en cuenta:

- La manera de obtener una referencia a un objeto remoto.
- El manejo de excepciones específicas de la invocación de métodos remotos.

Por su parte, un esqueleto es el encargado de recoger los parámetros que recibe del sustituto a través de la red, invocar de manera local al objeto remoto y devolver el resultado de nuevo a través de la red al sustituto del objeto que realizó la invocación (ver Figura 1.6).

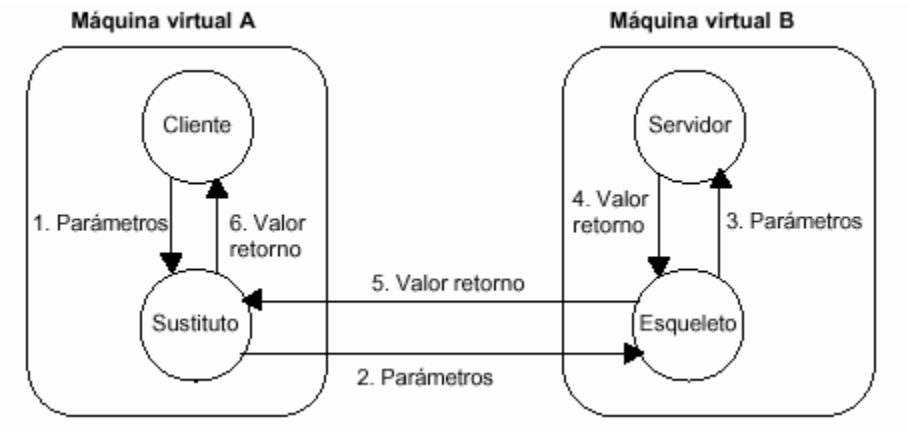

FIGURA 1.6 USO DE SUSTITUTOS Y ESQUELETOS EN LA INVOCACIÓN REMOTA.

A diferencia de una invocación local, una invocación RMI pasa los objetos locales que forman parte de la lista de parámetros por copia, que una referencia a un objeto local sólo sería útil en una máquina virtual única. RMI utiliza el **servicio de serialización de objetos** (proceso de leer y almacenar objetos, manejando el objeto como una secuencia de bytes, que podrá ser recuperado como el objeto original) para aplanar el estado de un objeto local y colocarlo en el mensaje a enviar a la máquina virtual remota. Si el objeto es no-serializable, no se puede usar como parámetro. También son pasados por valor los parámetros de tipos primitivos. Por otro lado, RMI pasa los objetos remotos (objetos que implementan una interfaz remota) por referencia.

1.5.3 Servicios

RMI proporciona interfaces y clases para buscar objetos remotos, cargarlos y ejecutarlos de manera segura. Adicionalmente, proporciona un **servicio de nombres** un tanto primitivo (servicio de nombrado no persistente y espacio de nombres plano) que permite localizar objetos remotos y obtener referencias a ellos, de manera que se puedan invocar sus métodos.

La utilización de objetos remotos lleva consigo la aparición de nuevas situaciones de error, de tal forma que el programador deberá manejar el conjunto de nuevas excepciones que pueden ocurrir durante una invocación remota. RMI incluye una característica de recolección de basura distribuida, que recolecta aquellos objetos servidores que no son referenciados por ningún cliente de la red.

1.5.4 Inconvenientes

- RMI sólo existe en el lenguaje Java. Por tanto, la interoperabilidad que ofrece entre objetos distribuidos se circunscribe únicamente a objetos Java, al contrario que CORBA o DCOM, por ejemplo.
- En el paso de parámetros, se mezclan dos semánticas, paso por valor y paso por referencia, que dependen del tipo de la entidad: objeto local, objeto remoto o tipo primitivo.

- La falta de transparencia se debe a que los objetos remotos se programan de manera diferente a los objetos locales, dado que tienen que implementar una interfaz determinada y heredar de una clase concreta que les permitan obtener la funcionalidad necesaria para ser invocados remotamente.

1.5 Conclusiones

Como se ha visto las herramientas para el desarrollo de cómputo distribuido tiene como inconveniente principal que trabajan sobre su área y aún no son del todo funcionales. Es por eso que ProActive PDC que es una biblioteca para cómputo paralelo, distribuido y concurrente orientado a objetos, hecho en Java, salva muchos de los inconvenientes de las herramientas anteriores:

- Orientado a objetos 100%
- Uniformidad en el paso de parámetros
- Programación transparente
- Facilita la migración, gracias a Java.
- Manejo de objetos activos autónomos
- Proporciona transparencia en el uso de RMI

No deja de ser un "*middlewar*e*"*, sin embargo son mayores las bondades, que lo hacen idóneo para el desarrollo de aplicaciones distribuidas y mas aún el desarrollo de las mismas con agentes móviles bajo un ambiente de ejecución heterogéneo y con una infraestructura genérica para la administración de los mismos. Las cualidades de ProActive se describirán en el siguiente capítulo.

Capítulo 2 ProActive PDC

2.1 Introducción

La biblioteca ProActive es un proyecto del consorcio ObjectWeb, el cual es una comunidad de Software Open Source middleware creada a final de 1999 por FRANCE TELECOM RD, Bull y por el INRIA (Instituto Nacional de Investigación en Informática y Automatización).

ProActive PDC es una biblioteca para cómputo paralelo, distribuido, concurrente orientado a objetos con seguridad y movilidad desarrollada en Java. Está biblioteca está hecha a partir de clases estándares de Java y proporciona una Interfaz de Programación de Aplicaciones (API) propia.

2.2 Principios

La biblioteca ProActive está basada en cuatro principios fundamentales, que se describen a continuación.

2.2.1 La transparencia entre la computación secuencial, multihilos y distribuida

La transparencia del modelo de programación implica la reutilización de código y transiciones suaves e incrementales entre la programación de cómputo secuencial, multihilos y distribuido, de tal forma que al programador se le simplifica la implementación de estos sistemas a través de primitivas simples. Por ejemplo en la Figura 2.1, muestra la ejecución de un programa secuencial utilizando un objeto activo, en la versión multihilos se distribuyen las operaciones utilizando varios objetos activos. Y para mayor rendimiento del sistema de computo estos objetos activos migran a diferentes computadoras de tal forma que el procesamiento sea mas eficiente (ver Figura 2.1).

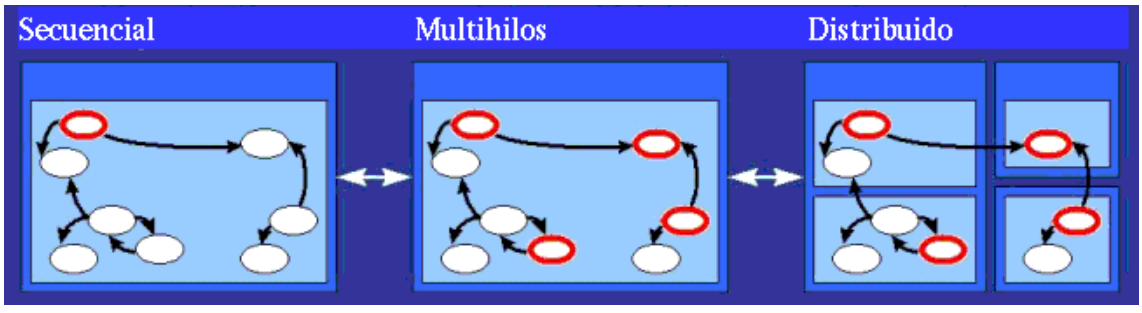

FIGURA 2.1 CÓMPUTO SECUENCIAL, MULTIHILOS Y DISTRIBUIDO

2.2.2 Objetos activos: unión de hilos y objetos remotos

Un objeto activo (OA) es un objeto[2] estándar al que se le ha proporcionado la habilidad para tener transparencia de localización, transparencia de actividad y sincronización, con lo anterior se refiere a que con código sencillo es posible conocer la ubicación actual del objeto, programar la actividad del objeto y establecer políticas de claras políticas de sincronización respectivamente.

Hay tres maneras de transformar un objeto estándar en un (OA):

1) Basado en clase
Object[] params = new Object[] {new Integer (26), "astring"};
A a = (A) ProActive.newActive ("ejemplo.A", params, node);

2) Basado en instanciación
public class AA extends A implements Active {}
Object[] params = new Object[] { new Integer(26), "astring" };
A a = (A) ProActive.newActive("ejemplo.AA", params, node);

3) Basado en objeto
Permite cambiar un objeto estándar a un objeto activo y remoto, para el cual no se tiene el código fuente (característica necesaria en el contexto de código móvil).
A a = new A (26, "astring");
a = (A) ProActive.turnActive (a, node);

2.2.3 El modelo de cómputo

Este modelo está basado en trabajos y estudios descritos [CKV98] y esta compuesto de las siguientes características:

- **Modelo heterogéneo,** uso de objetos activos como objetos pasivos (estándar).
- **La comunicación sistemática asíncrona**, esto permite un procesamiento concurrente, por ejemplo el objeto1 invoca un método en el objeto2, este método devuelve un resultado en un componente. El objeto1 obtiene un objeto futuro (OF, objeto eventual, contenedor de resultados) y puede continuar su tarea hasta que realmente use el resultado de la llamada al método.
- **Objetos pasivos no compartidos,** utiliza la llamada por valor entre objetos activos.
- **Continuaciones automáticas,** mecanismo de delegación transparente que permite pasar como parámetro o resultado de un objeto futuro sin bloquearse por esperar el resultado. Cuando el resultado esta disponible en el objeto que origino la creación del objeto futuro, este objeto debe actualizar el resultado en todos los objetos donde el resultado fue puesto.
- **Espera-por-necesidad** (*"wait-by-necessity"*), es una política de sincronización de objetos futuros de forma automática y transparente.

[2] Instancia de una clase

- **Control centralizado y explicito**, hace uso de las bibliotecas de abstracciones (conjunto de primitivas) .

Una condicionante en el diseño del modelo consiste en habilitar el uso de precondiciones, postcondiciones (asertos que contienen una expresión booleana) e invariantes (condiciones que siempre se cumplen en determinadas líneas de código), para la evaluación de código que garanticen un determinado comportamiento.

2.2.4 La reutilización y transparencia

Se promueve la reutilización de código existente con el fin ahorrar tiempo en la programación y en ProActive se cuenta con dos aspectos que permiten tal reutilización.

1) *Espera-por-necesidad*: política de sincronización entre objetos, objetos futuros transparentes implícitos y sistemáticos, facilidad de programación sincronización y la reutilización de métodos existentes.

2) Polimorfismo entre objetos activos y estándar
- tipo de compatibilidad para clases y no solamente para interfases.
- Necesario y aplicado para los objetos futuros
- Mecanismo dinámico

2.3 La base de ProActive

La biblioteca ProActive esta basada en objetos activos, éstos define a la biblioteca como tal.

2.3.1 El Objeto activo

Los objetos activos son unidades básicas de actividad y distribución, estos son usados para construir aplicaciones concurrentes. Un objeto activo posee su propio hilo, éste hilo ejecuta solamente los métodos invocados en dicho objeto activo por otros objetos activos y pasivos del subsistema al que pertenece este objeto activo.

La diferencia de Java estándar y ProActive PDC es que el programador no tiene que manipular explícitamente los hilos. Los objetos activos pueden ser creados en cualquiera de los anfitriones involucrados en las computadoras participantes. Una vez que un objeto activo es creado, su actividad (el hecho de que éste tenga su propio hilo) y su localización (local o remota) son perfectamente transparentes, de hecho cualquier objeto activo puede ser manipulado tal y como si este fuera una instancia pasiva.

2.3.2 Modelo propuesto por Eiffel//

Las principales características de este modelo el cual esta detallado en [CKV98] son:

- *La aplicación está estructurada en subsistemas.*

Hay un objeto activo (y por lo tanto un hilo) para cada subsistema y un subsistema para cada objeto activo (o hilo). Cada subsistema está entonces compuesto de un objeto

activo y de cualquier número de objetos pasivos (posiblemente ninguno). El hilo de un subsistema solamente ejecuta métodos en los objetos de este subsistema.

- *No hay objetos pasivos compartidos entre los subsistemas*

Estas dos características tienen consecuencias de gran importancia en la topología de la aplicación, estas son:

  - De los objetos que componen a un subsistema (objetos activos y pasivos), solamente el objeto activo es conocido por los objetos que se encuentran fuera del subsistema.
  - Todos los objetos (pasivos y activos) pueden tener referencias dentro de objetos activos.
  - Si un objeto O1 tiene una referencia dentro de un objeto pasivo O2, entonces O1 y O2 son parte del mismo subsistema.

Esto también afecta la semántica en el paso de mensajes entre subsistemas:

- Cuando un objeto en un subsistema llama a un método en un objeto activo, el parámetro de la llamada debe hacer referencia en objetos pasivos del subsistema (ver Figura 2.2), lo cual conlleva a objetos pasivos compartidos. Este es el por qué de que objetos pasivos sean pasados como parámetros de llamadas en objetos activos por copia profunda[3] (*deep copy*). Los objetos activos, por otro lado son siempre pasados por referencia. Simétricamente, esto también se aplica para objetos que son devueltos de métodos invocados en objetos activos.

- Cuando un método es llamado en un objeto activo, éste regresa inmediatamente (como el hilo no puede ejecutar métodos en otro subsistema) un objeto futuro. Un objeto futuro, es un contenedor para el resultado de la invocación al método, cuando este es devuelto. Desde el punto de vista del subsistema invocador no hay diferencia alguna entre el objeto futuro y el objeto que podría haber sido regresado si la misma llamada hubiera sido hecha dentro de un objeto pasivo. Entonces el hilo que invoca puede continuar ejecutando su código tal y como si la llamada hubiera sido efectivamente realizada. El rol del objeto futuro es bloquear este hilo si éste invoca a un método en el objeto futuro y el resultado aún no a sido obtenido (ejemplo: el hilo del subsistema en el cual la llamada fue recibida no ha ejecutado la llamada y puesto el resultado en el objeto futuro) esta política de sincronización entre objetos es conocida como **espera por necesidad.**

[3] es la clonación de un objeto donde se copia con todos sus agregados y compuestos

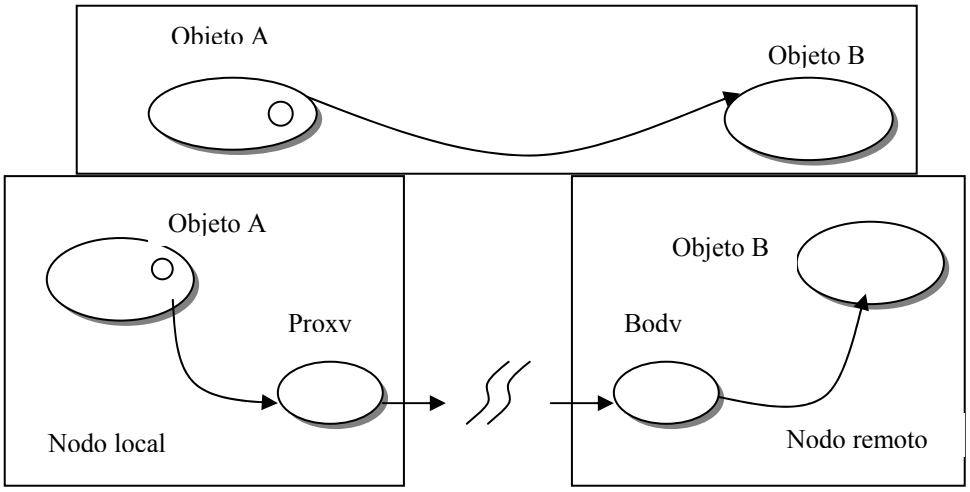

FIGURA 2.2 COMPARACIÓN DE LLAMADAS EN UN OP COMO CONTRARIO A UN OA

2.3.3 Composición de un objeto activo

El objeto activo es en realidad: Un cuerpo y un objeto Java estándar. El cuerpo no es visible desde fuera del objeto activo. Entonces todo luce tal y como si el objeto activo fuera estándar. El cuerpo es responsable de recibir llamadas en el objeto activo, almacenando estas llamadas en una cola de llamadas pendientes (que también se llaman peticiones), éste también ejecuta estas llamadas en un orden dado por una política de sincronización especifica, si ninguna política de sincronización específica se proporciona, las llamadas son manejadas de manera FIFO, entonces el hilo de un objeto activo escoge alternativamente un método en la cola de peticiones pendientes y la ejecuta.

Cuando el sistema envía una llamada a un objeto activo, este objeto activo es representado por un *proxy*, cuya responsabilidad principal es generar objetos futuros para la representación de valores futuros, transformando llamadas en objetos *Request* (en términos de metaobjetos, esto es una abstracción) y ejecuta una copia profunda de objetos pasivos pasados como parámetros.

2.4 Objetos activos: creación y conceptos avanzados

2.4.1 Creación de un objeto activo

En una aplicación puede haber tanto instancias pasivas como activas de una clase dada. Aunque casi cualquier objeto se puede convertir en un objeto activo hay algunas restricciones que serán detalladas posteriormente, en la Figura 2.3 .

La forma de crear una instancia pasiva de A es:

A a = new A(26,"astring");

ProActive hay dos maneras de crear un objeto activo:

1) Una forma es usar **ProActive.newActive** y esta basado en la instanciación de un nuevo objeto.

2) Y la otra es usar **ProActive.turnActive** y esta basado en el uso de un objeto existente.

```
Public class A implements Active{};
Object[ ] params = new Object[ ]
{
  new Integer (26), "astring");
  try {
        a= (A) ……;
        } catch (ActiveObjectCreationException e)
                { // falló la creación del objeto activo
                 e.printStackTrace();
                }
}
```

FIGURA 2.3 CLASE A

Creación basada en instanciación

Cuando se usa la creación basada en instanciación cualquier argumento pasado al constructor del objeto materializado (proceso de convertir un objeto abstracto a un objeto activo) a través de **ProActive.newActive** es serializado y pasado por copia al objeto. El objeto activo puede ser instanciado local o remotamente, garantizándose que los parámetros siempre van hacer pasados por copia al constructor. Por otro lado se debe asegurar que los argumentos del constructor son serializables. Sin embargo la clase usada para crear el objeto activo **no necesita ser** serializable, aún en el caso en que el objeto activo sea creado remotamente.

```
 A a;
Object[ ] params = new Object[ ]
{
  new Integer (26), "astring");
  try {
        a= (A) ProActive.newActive("example.A", params);
        } catch (ActiveObjectCreationException e)
                { // falló la creación del objeto activo
                 e.printStackTrace();
                }
        catch (NodeException ex) {
            ex..printStackTrace();
          }
}
```

FIGURA 2.4 CREACIÓN DE UN OA POR INSTANCIACIÓN

El código que se muestra en la Figura 2.4, crea un objeto activo de la clase **A** en la máquina virtual de Java (MVJ) local. Si la invocación del constructor de la clase **A** lanza una excepción, ésta cae dentro del tipo de excepciones **ActiveObjectCreationException.** Cuando la llamada a *newActive* regresa, el objeto activo ha sido creado y su hilo activo se ha inicializado.

El primer parámetro de *newActive* es una cadena que contiene el nombre totalmente válido de la clase que se desea hacer activa.

Los parámetros del constructor son pasados como un arreglo de objetos, entonces de acuerdo al tipo de elementos de este arreglo, en el tiempo de ejecución de ProActive se determina que constructor de clase **A** se llamará. Sin embargo esto provoca ambigüedad en la resolución del constructor debido a que:

a) Como los argumentos del constructor son almacenados en un arreglo de tipo **Object[],** los tipos primitivos tienen que ser representados por sus tipos de empaquetamiento de objetos. En la Figura 2.4, se usa el objeto **Integer** para empaquetar el valor entero *int 26*. La ambigüedad surge si dos constructores de la misma clase solamente difieren por la conversión de los tipos primitivos a su correspondiente clase empaquetadora.

b) Si un argumento es nulo, en el tiempo de ejecución obviamente no es posible determinar su tipo. Otra ambigüedad surge en la Figura 2.5, entre el tercer y cuarto constructor si el segundo elemento del arreglo es nulo.

```
public A( int i) {   // }
public A( Integer  i {  // }
public A( int i, String s) {  // }
public A( int i, Vector v) {  // }
```

FIGURA 2.5 CONSTRUCTORES CON DIFERENTES PARÁMETROS

Es posible pasar un tercer parámetro a *newActive* para crear el nuevo objeto activo en una MVJ específica, posiblemente remota. La MVJ es identificada usando un objeto **Node** que ofrece los servicios mínimos que ProActive necesita en una MVJ para comunicarse con ella. Si ese parámetro no se da, el objeto activo se crea en la MVJ local y es asignado al nodo por defecto.

Un nodo es identificado por el URL, el cual se forma usando el protocolo, con los siguientes datos: el nombre de la máquina anfitriona donde reside la MVJ y donde esta ubicado el nodo y el nombre del nodo. El **NodeFactory** permite crear o buscar nodos. El método *newActive* puede tomar como parámetro el URL del nodo como una cadena o como un objeto **Node** que apunta a un nodo existente, la Figura 2.6 muestra un ejemplo de ello.

```
A = (A) ProActive.newActive("example.A",params,
"rmi://pluto.inria.fr/aNode");
      or
Node node = NodeFactory.getNode("rmi://pluto.inria.fr/aNode");
A = (A) ProActive.newActive("example.A", params,node);
```

FIGURA 2.6 CREACIÓN DE UN OBJETO ACTIVO EN UN URL DADO

Creación basada en objetos

Este tipo de creación se usa para convertir un objeto pasivo existente a uno activo, lo anterior se presenta en ProActive como respuesta al siguiente problema:

Considerando, por ejemplo que una instancia de la clase **A** se crea dentro de una biblioteca y se devuelve como el resultado de una llamada a un método. Esto trae como consecuencia, que no se tenga acceso al código fuente donde el objeto fue creado, lo cual previene de modificarlo para crear una instancia activa de **A**. Aún si esto fuera posible, esto no es posible ya que no se desea una instancia activa de **A** por cada llamada en este método.

Cuando se usa la creación basada en objetos, se crea el objeto que esta siendo materializado como un objeto activo antes del manejo, por lo tanto no hay serialización cuando se crea el objeto.

Cuanto se invoca **ProActive.turnActive** en el objeto, dos casos se pueden presentar, si se crea el objeto activo localmente (en un nodo local), este no será serializado, si se crea el objeto activo de manera remota (en un nodo remoto), el objeto materializado si será serializado. Por lo tanto si *turnActive* es aplicado en un nodo remoto, la clase usada para crear el objeto activo de esta forma **tiene que ser** *Serializable.* En suma, cuando se usa *turnActive* se debe de tener mucho cuidado de que no se guarden las referencias en objeto original después de la llamada a *turnActive*. Una llamada directa al método del objeto original, sin pasar por un sustituto de ProActive en éste objeto rompería el modelo. La Figura 2.7 muestra el código para la creación basada en objetos

```
A a  = new A (26,"astring");
   a = (A) ProActive.turnActive(a);
```

FIGURA 2.7 CREACIÓN DE UN OBJETO ACTIVO BASADO EN OBJETOS

Tal como en *newActive* y el segundo parámetro de *turnActive* sí se dan, indican la posición del objeto activo a ser creado, sin parámetros o nulo significa que el objeto activo será creado localmente en el nodo actual. Cuando se usen estos métodos, el programador se debe asegurar que no exista ninguna otra referencia en el objeto pasivo *a* después de la llamada a *turnActive*. Si tales referencias fueran usadas para llamar métodos de manera

directa en objeto pasivo *a* (sin pasar por cuerpo), el modelo se volvería inconsistente y por lo tanto no se podría garantizar la sincronización.

2.5 La actividad de un objeto activo

Personalizar la actividad del objeto activo es una característica muy particular de ProActive debido a que éste permite especificar completamente el comportamiento de un objeto activo. Por defecto un objeto que se convierte en un objeto activo atiende las peticiones entrantes de manera FIFO (First In-First Out, primero en entrar, primero en salir). Para especificar alguna otra política para atender las peticiones ó para especificar cualquier otro comportamiento se puede implementar interfaces definiendo métodos que serán llamados automáticamente por ProActive.

Es posible especificar que hacer antes de que la actividad inicie, la actividad misma y que hacer al final de la misma. Los pasos son:

1. la inicialización de la actividad (se hace solamente una vez)
2. la actividad misma
3. el final de la actividad (se hace solamente una vez)

Se usan tres interfaces para definir e implementar cada paso:

1. InitActive
2. RunActive
3. EndActive

En caso de una migración, un objeto activo se detiene y reinicia su actividad automáticamente sin invocar las fases de inicialización o finalización. Solo la actividad se reinicializa por sí misma.

Existen dos formas para definir cada una de las tres fases de un objeto activo:

1. Implementando uno o más de las tres interfases directamente en la clase usada para crear el objeto activo.
2. Pasando un objeto, implementando uno o más de las tres interfaces en el parámetro al método *newActive* ó *turnActive.*

Note que los métodos definidos por estas tres interfaces, garantizan que serán invocadas por el hilo activo del objeto activo.

Algoritmos para decidir que actividad invocar, los algoritmos que deciden que hacer en cada fase son los siguientes:
(Nota: **activity** es el objeto eventual que se pasa como parámetro a *newActive* o *turnActive*)

InitActive
Si la actividad es no nula e implementa **InitActive**
entonces
se invoca el método **InitActivity** definido en la actividad del objeto
si no
si la clase del objeto materializado implementa **InitActive**
entonces
se invoca el método **InitActivity** del objeto materializado
si no
no se realiza ninguna inicialización

RunActive
Si la actividad es no nula e implementa **RunActive**
entonces
se invoca el método **runActivity** definido en la actividad del objeto
si no
si la clase del objeto referenciado implementa **RunActive**
entonces
se invoca el método **runActivity** del objeto referenciado
si no
se ejecuta la actividad estándar FIFO

EndActive
si la actividad es no nula e implementa **EndActive**
entonces
se invoca el método **endActivity** definido en la actividad del objeto
si no
si la clase del objeto implementa **EndActive**
entonces
se invoca el método **endActivity** del objeto materializado
si no
no se realiza ninguna limpieza.

La implementación de las interfaces directamente en la clase usada para crear el objeto activo, es la solución más fácil, cuando se controla la clase que se desea hacer activa, dependiendo en que fase de la vida del objeto activo se requiera personalizar, se implementa la correspondiente interfase (una o más) entre **InitActive**, **RunActive** y **EndActive**.

El ejemplo que tiene una inicialización y una actividad personalizada se puede ver en la Figura 2.8.

```
import org.objectweb.proactive.*;

public class A implements InitActive, RunActive
{
     private String myName;
     public String getName() {  return myName;}
    //--Implementacion de InitActive
   public void initActivity(Body body)
        { myName = body.getName();}
   //--Implementación de RunActive para servir peticiones en orden FIFO
   public void runActivity(Body body)
      {
        Service service = new Service(Body);
        while (body.isActive()) {
              sService.blokingServeYoungest();
           }
      }

public static void main (string {} args) throws Exception
  {
    A a = (A) ProActive.newActive("A",null);
    System.out.println("Name = "+a.getName());
  }
}
```

La solución cuando no se tiene un control de la clase que se desea hacer activa ó cuando se requiere escribir políticas de actividades genéricas y reutilizarlas con diversos objetos activos, es pasar un objeto con las interfases implementadas en el momento que se crea el objeto activo.

Dependiendo en que fase de la vida del objeto activo se requiera personalizar, se implementa la correspondiente interfaz (una o más) entre **InitActive**, **RunActive** y **EndActive**. En la Figura 2.9 se personaliza la actividad para después ser pasado como parámetro durante la creación del objeto activo.

```
import org.objectweb.proactive.*;
public class LIFOActivity implements RunActive {
      //implementación de RunActive para servir peticiones de forma LIFO
      public void runActivity (Body body) {
         Service service = new Service(body);
         while (body.isActive()) {
              service.blockingServeYoungest();
          }
      }
}
import org.objectweb.proactive.*;
public class A  implements InitActive {
   private String myName;
}
// implements InitActive
public void initActivity (body body) {
    myName = body.getName();
}
public static void main (string[] args) throws Exception {
    //newActive (nombre d e la clase, parámetro del constructor (null = none),
   //node (null =local), active, MetaObjectFactory (null = default)
  A a = (A) ProActive.newActive("A",null,null,new LIFOActivity(),null);
  System.out.printl("Name = "+ a.getName());
 }
}
```

Figura 2.9 Interfase implementada pasada en la creación de un OA

Comparando la solución de la Figura 2.8 donde las interfaces se implementan directamente en la clase materializada con la solución de la Figura 2.9, en esta última surge la restricción de que no es posible acceder el estado interno del objeto materializado. Un objeto externo se debe usar cuando no exista la necesidad de acceder a las variables miembro del objeto materializado y su actividad implementada sea suficientemente genérica.

2.6 Restricciones en los objetos materializados

No todas las clases pueden generar objetos activos. Existen algunas restricciones, la mayoría de ellas son causadas por la compatibilidad con java al 100%, la cual prohíbe modificar la MVJ o el compilador.

Algunas de estas restricciones trabajan a nivel de clase por ejemplo:

- Las clases **Final** no pueden generar objetos activos
- Las clases no publicas también no pueden generar objetos activos
- Las clases sin un constructor sin argumento no pueden ser materializadas.

Algunas otras restricciones pasan a nivel de un método en una clase específica por ejemplo:

- Los métodos **Final** no pueden ser usados del todo. Cuando se invoca un método final en un objeto activo esto causa un comportamiento inconsistente.
- La invocación de un método no público en un objeto activo levanta una excepción. (esta restricción desapareció con la versión 1.2 de JDK)

2.8 El patrón de diseño de fábrica (factory)

La creación de un objeto activo usando ProActive puede ser un poco incomodo y requerir mas líneas de código que para crear un objeto regular. Una solución a este problema es el uso del patrón de diseño denominado **factory**. Este se aplica principalmente a la creación basada en clases. Este consiste en adicionar un método estático a la clase **AA** que cuida de instanciar el objeto activo y regresarlo. El código puede ser visto la Figura 2.10.

```
public class AA extends A
{
   public static A createActiveA (init I, String s, Node node)
  {
  object{} params = new Object{} {new Integer (i), s};
  try {
       return (A) ProActive.newActive("A",params, node);
     }catch (Exception e)  {
                              System.err.println ("la creación de una instacia de A a levantado
                              una excepcion: "+e);
                              return null;
                             }
  }
}
```

FIGURA 2.10 APLICACIÓN DEL PATRÓN DE DISEÑO FACTORY

Es responsabilidad del programador decidir si este método tiene que lanzar excepciones o no. Se recomienda que este método solamente maneje las excepciones que aparecen en la firma del constructor materializado. Pero las excepciones no funcionales inducidas por la creación del objeto activo tienen que ser repartidas en algún lugar en el código.

2.9 Conceptos avanzados

2.9.1 Personalizando el cuerpo del objeto activo

Hay muchos casos donde es deseable especificar completamente el comportamiento de un objeto activo, es decir agregar nuevas funciones para un uso particular, por ejemplo cuando se esta creando un objeto activo, se podría requerir agregar algunos mensajes de depuración ó algunos comportamientos con cuestiones del tiempo en el envío o recepción de peticiones. Pero el cuerpo no es un objeto modificable, éste delega la mayoría de sus tareas en objetos que lo apoyan llamados metaobjetos. Los metaobjetos estándar son

usados por defecto en ProActive pero es posible reemplazar cualquiera de estos metaobjetos por uno personalizado.

Se ha definido la interfase **MetaObjectFactory** capaz de crear las fábricas para cada uno de esos metaobjetos Esta interfase está implementada por **ProActiveMetaObjectFactory** que proporciona todas las fábricas por defecto usadas en ProActive.

Cuando se crea un objeto activo como se mencionó anteriormente, es posible especificar cual **MetaObjectFactory** usar para cada instancia en particular del objeto activo que esta siendo creado.

Para personalizar, primero se debe escribir una nueva fábrica **MetaObject** que herede de **ProActiveMetaObjectFactory** ó implementar directamente **MetaObjectFactory** para redefinir cada cosa. Heredando de **ProActiveMetaObjectFactory** se ahorra tiempo de tal manera que solo se tendría que redefinir lo que realmente se necesita. Un ejemplo se muestra en la Figura 2.11:

```
public class MyMetaobjectFactory extends ProActiveMetaObjectFactory {
   private static final MetaObjectfactory instance = new MyMetaobjectFactory();
  protected MyMetaobjectFactory() {
      super();
  }
public static MyMetaobjectFactory newInstance()  {
     return instance;
}

//Métodos protegidos

protected RequestFactory newRequestFactorySingleton() {
    return new myRequestFactory();
}

//Clase Interior

protected class MyMetaobjectFactory implements RequestFactory, java.io.Serializable {
public Request newRequest (MethodCall methodCall, UniversalBody sourceBody, boolean
isOneWay, long sequenceID) {
   return new MyRequest (methodCall, sourceBody, isOneWay, sequenceId, server);
}
}// fin de las clases interiores
}
```

FIGURA 2.11 USO DE PROACTIVEMETAOBJECTFACTORY

La fabrica de la Figura 2.11, simplemente redefine el **RequestFactory** para hecer que el cuerpo use un nuevo tipo de petición. El método **protegido RequestFactory**

newRequestFactorySingleton() es un método conveniente que usa **ProactiveMetaObjectFactory** para simplificar la creación de las fabricas como singleton[4].

El uso de esta nueva fábrica es bastante simple, todo lo que se tiene que hacer es pasar una instancia de la fábrica cuando esta creando un nuevo OA, si retomamos el mismo ejemplo anterior tenemos el código en la Figura 2.12:

```
Object[] params = new Object[] {new Integer (26), "astring"};
 try {
          A a = (A) ProActive.newActive("example.AA", params, null, null,
          MyMetaObjectFactory .newInstance());
     }catch (Exception e) {
                                    e.printStackTrace();
                           }
```

FIGURA 2.12 PASO DE LA INSTANCIA DE FÁBRICA EN EL OA

En el caso de un **turnActive** el código debe ser:

A a = new A(26, "astring");
a = (A) ProActive.turnActive(a,null,null,MyMetaObjectFactory.newInstance());

2.9.2 El rol de los componentes de un objeto activo

En esta sección se tendrá una visión cercana de lo que sucede cuando un OA es creado, proporcionando un mejor entendimiento de cómo trabaja la biblioteca y de donde vienen las restricciones de ProActive.

En alguna parte del código de una instancia de la clase **A**, se crea un OA de la clase **B,** usando el código de la Figura 2.13:

```
B b;
 Object{} params = {};
 try {
       //se crea una instancia activa de
       / B en el nodo actual
       b = (B) ProActive.new.Active ("B",params);
     } catch (Exception e) {
       e.printStackTrace();
       }
```

FIGURA 2.13 CREACIÓN DEL OBJETO ACTIVO B

[11] clase que solo tiene una instancia

Si la creación de la instancia activa de **B,** es satisfactoria, el grafo de objetos es como se describe en la Figura 2.14 (con flechas que denotan las referencias).

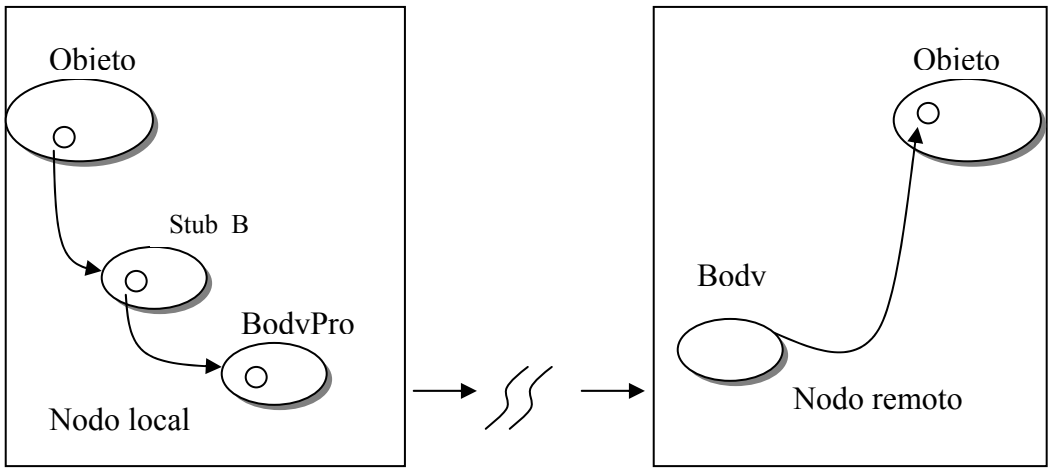

FIGURA 2.14 LOS COMPONENTES DE UN OBJETO ACTIVO

La instancia activa de B está compuesta de 4 objetos:

- Un Stub (Stub_B)
- Un proxy (BodyProxy)
- Un cuerpo (Body)
- Una instancia de B

El rol de la clase **Stub_B** es abstraer todas las llamadas al método que puedan ser ejecutadas a través de una referencia de tipo **B** y solamente éstas como llamadas a métodos declarados en una subclase de **B,** por downcasting[5] causarían un error en tiempo de ejecución. Abstraer una llamada simplemente significa construir un objeto (en este caso, todas las llamadas materializadas son instancias de la clase **MethodCall**) que representa la llamada, de tal manera que si esta puede ser manipulada como cualquier otro objeto. Esta llamada materializada es entonces procesada por los otros componentes del OA para obtener el comportamiento que esperamos de un OA.

La idea de usar un objeto estándar para representar elementos del lenguaje que normalmente no son objetos (tales como llamadas de métodos, llamadas del constructor (referencias, tipos, etc.), esta relacionada con la *programación metaobjeto*. El protocolo metaobjeto (MOP), esta construido como se describe en [MOP] pero no es un prerrequisito para entender y usar ProActive.

Como uno de los objetivos es proporcionar OA transparentes, las referencias a un OA de la clase **B** necesitan ser del mismo tipo que las referencias a instancias pasivas de **B** (esta característica es llamada *polimórfismo* entre instancias pasivas y activas de la misma clase). Esto es, por construcción, **Stub_B** es una subclase de la clase **B**, permitiendo por lo tanto que instancias de la clase **Stub_B** sea asignado a variables de tipo **B**.

[5] conversión del tipo padre al verdadero tipo del objeto

La clase **Stub_B** redefine cada uno de los métodos heredados de su superclase. El código para cada método de la clase **Stub_B** construye una instancia de la clase **MethodCall** para representar la llamada a este método. Este objeto es entonces pasado al **BodyProxy**, el cual devuelve un objeto que fue devuelto como el resultado de la llamada al método.

Desde el punto de vista del invocador, cada cosa luce tal como si la llamada hubiera sido ejecutada en una instancia de B.

Ahora que se sabe como trabajan los sustitutos, se debe entender algunas de las limitaciones de ProActive:

- Obviamente, **Stub_B** no puede redefinir los métodos de tipo **final** heredados de la clase **B**. Por lo tanto, las llamadas a esos métodos no son materializadas sino más bien ejecutadas en el **stub**, lo cual puede llevar a comportamientos inexplicables si el programador no evita cuidadosamente el llamado a los métodos **final** en objetos activos.

  Hay seis métodos finales en la clase base **Object**, 5 de los 6 métodos tienen que lidiar con la sincronización entre hilos (**notify(), notifyAll()** y 3 versiones de **wait()**). Estos métodos no deberían ser usados dado que un OA proporciona sincronización de hilos.

  Si se usa el mecanismo de sincronización hilos estándar y el mecanismo de sincronización de hilos de ProActive al mismo tiempo puede traer conflictos y como consecuencia y dificultar el proceso de depuración.

  El método 6 de tipo **final** de la clase Object es **getClass()**. Cuando se invoca en un OA, **getClass()** no es materializado y por lo tanto ejecutado en el objeto stub, el cual devuelve un objeto de clase **Class** que representa la clase del stub (**Stub_B** en el ejemplo) y no de la clase del OA (**B** es el ejemplo). Sin embargo este método raras veces es usado en aplicaciones estándar y no previene el operador **instanceof** para trabajar gracias a su comportamiento polimórfico. Por lo tanto la expresión (**foo instanceof B**) tiene el mismo valor cuando **B** es un objeto activo y cuando no lo es.

- La obtención y la puesta de variables de instancias de manera directa (no a través de un método que lo obtenga o lo coloque) debe ser evitado en el caso de los OA debido a que éste podría resultar en obtener o colocar un valor en el objeto sustituto y no en la instancia de la clase **B**. Este problema es normalmente atacado usando los métodos get/set para colocar o leer atributos. Esta estricta regla de encapsulación, puede ser encontrada en JavaBean ó en los sistemas distribuidos de objetos RMI o Corba.

  El rol del proxy: es manejar el asincronismo en llamadas a objetos activos. Mas específicamente, éste crea objetos futuros si van a ser necesarios, llama a los cuerpos y objetos futuros para los sustitutos. Como esta clase opera en objetos

MethodCall, éste es absolutamente genérico y no depende de todos en el tipo del sustituto que alimenta las llamadas a través de su método **reify.**
El papel del cuerpo: es el de almacenar las llamadas (objetos **Request**) en una cola de peticiones pendientes y procesarlas de acuerdo a una política de sincronización dada, cuyo comportamiento por defecto es el FIFO. El cuerpo tiene su propio hilo, que alternativamente selecciona una petición en la cola de peticiones pendientes y ejecuta la llamada asociada.

El rol de la instancia de la clase B: contiene información de sincronización en su método **live,** si es que hubiera alguna. Como el cuerpo ejecuta llamadas una por una, no puede haber ninguna ejecución concurrente de dos porciones de código de este objeto por dos hilos diferentes.

Como una consecuencia, el uso de la palabra reservada **synchronized** en la clase **B** no es necesario. Cualquier esquema de sincronización que pueda ser expresada a través de monitores y sentencias **synchronized** pueden ser expresadas usando mecanismos de sincronización de alto nivel de ProActive de una manera más natural y amigable.

2.10 Objetos futuros y llamadas asíncronas

2.10.1 Creación de un objeto futuro

Siempre que sea posible, una llamada a un método de un objeto activo, se abstrae como una petición asíncrona. Si esto no es posible, la llamada es síncrona y se bloquea hasta que la respuesta sea recibida. En caso de que la petición sea asíncrona, ésta inmediatamente devuelve un OF.

Este objeto futuro actúa como un espacio de almacenamiento para el resultado de la invocación al método que aún no concluye. Como consecuencia, el hilo invocador puede continuar ejecutando su código, mientras esto sucede no tiene que invocar métodos sobre el objeto devuelto, en tal caso el hilo que invoca es automáticamente bloqueado sí el resultado de la invocación al método aún no esta disponible.

En la tabla 2.1 se muestran los diferentes casos en que se puede permitir una llamada asíncrona.

Tipo devuelto	Pasa una excepción	Creación de un objeto futuro	Asíncrono
Void	-	No	Si
Non Reifiable	-	No	No
Objetc	Si	No	No
Reifiable Object	No	Si	Si

TABLA 2.1 CASOS PARA PERMITIR UNA LLAMADA ASÍNCRONA.

La creación de un objeto futuro no solo depende del tipo del invocador, sino también del tipo del objeto devuelto. La creación de un objeto futuro solo es posible si el objeto es materializable.

Note que aunque éste tiene una estructura bastante similar a un objeto activo, un OF no es activo. Este solo tiene un sustituto y un proxy como se muestra en la Figura 2.15.

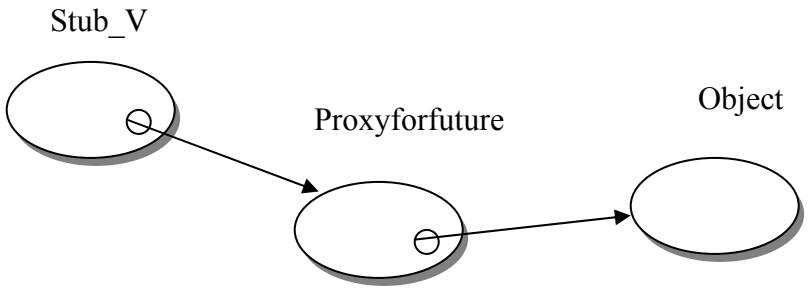

FIGURA 2.15 COMPOSICIÓN DE UN OBJETO FUTURO

Durante su tiempo de vida, un objeto activo puede crear muchos objetos futuros, estos se guardan automáticamente en un espacio de almacenamiento para objetos futuros (FuturePool).

Cada vez que un objeto futuro es creado, se inserta en el espacio de almacenamiento para OF del objeto activo correspondiente. Cuando el resultado llega a estar disponible, el OF es removido del espacio de almacenamiento. Aunque la mayoría de los métodos del FuturePool son solo para uso interno y son llamados directamente por la biblioteca ProActive, se proporciona un método para que espere hasta que el resultado este disponible. En lugar de bloquear hasta que un futuro especifico esté disponible, la llamada a **waitForReply(),** se bloquea hasta que cualquiera de los futuros actuales llegue a estar disponible.

Cualquier llamada a un OF es materializada para ser bloqueada si el objeto futuro no está aún disponible y después ejecutada sobre el objeto resultado. Sin embargo, existen dos métodos que no siguen el mismo esquema: Equals y HashCode. Ellos a menudo son llamados por otros métodos desde la biblioteca de Java, como HashTable.add() y así están mas fuera del control del usuario la mayoría de las veces. Esto puede conducir fácilmente a candados mortales si ellos son llamados en un objeto que no esta aún disponible.

En lugar de regresar el código hash del objeto, *hashCode()* devuelve el código hash de su proxy. Debido a que hay solo un proxy por OF, hay una única equivalencia entre ellos.

La implementación por defecto de equals() en la clase Object es para comparar las referencias en dos objetos. En ProActive, se redefine para comparar el código hash de dos proxys. Como consecuencia solo es posible comparar dos objetos futuros y no un OF con un objeto normal.

Con esta técnica se presenta la desventaja de que el usuario no va a poder anular los métodos por defecto de hashcode() y equals().

toString( ): el método toString es llamado con **System.out.println()** para cambiar un objeto a una cadena imprimible. En la implementación, una llamada a este método bloqueará al OF como cualquier otra llamada, así que hay que tener cuidado en su uso. Por ejemplo, el intentar imprimir un OF con el propósito de depurar, conducirá a un candado mortal. En lugar de desplegar la cadena correspondiente de un OF, se debería considerar desplegar su código hash.

2.10.2 Llamadas asíncronas en detalle

Para describir las llamadas asíncronas se basará en el siguiente ejemplo en donde algunas piezas de código en una instancia de la clase **A** llaman al método **foo** en una instancia activa de la clase **B**. Esta llamada es asíncrona y devuelve un objeto futuro de la clase **V**. Entonces, después de haber ejecutado algún otro código, el mismo hilo que emitió la llamada, llama al método **bar** en el **OF** devuelto por la llamada a **foo**. Este es un comportamiento secuencial modelado en el diagrama de secuencia de la Figura 2.16, note como un solo hilo sucesivamente ejecuta código de diferentes métodos en diferentes clases.

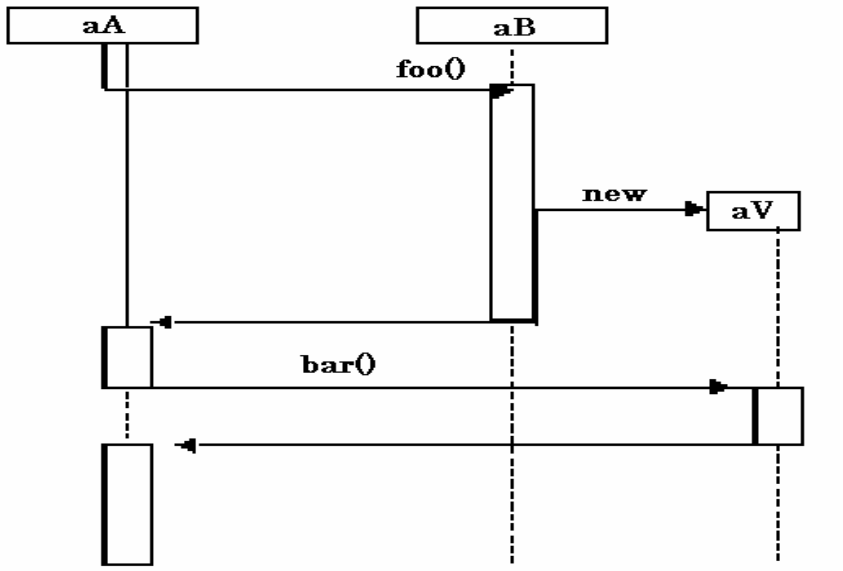

FIGURA 2.16 DIAGRAMA DE SECUENCIA DE LA EJECUCIÓN SECUENCIAL

Se presenta un grafo de la ejecución de los objetos en tres momentos diferentes de la ejecución:

- Antes de llamar a **foo**, se tiene exactamente la misma configuración como después de la creación de la instancia activa de **B**, tal y como se muestra en la Figura 2.17, una instancia de la clase **A** y una instancia activa de la clase **B**. Como todos los OA, la instancia de la clase **B** esta compuesta de un esqueleto (una instancia de la clase **Stub_B**, la cual hereda directamente de **B**), un **BodyProxy**, un **Body** y la instancia actual de **B.**

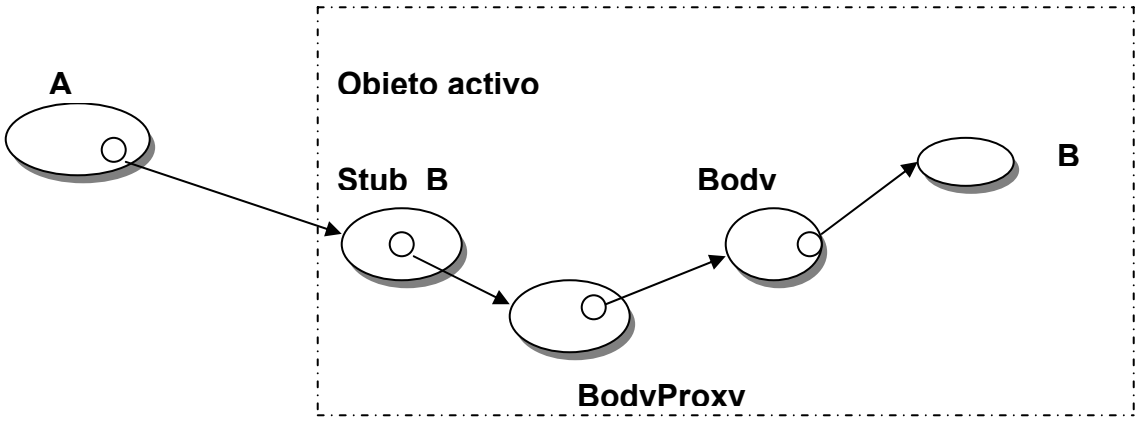

FIGURA 2.17 LOS COMPONENTES DE UN OBJETO ACTIVO

- Después de que la llamada asíncrona a **foo,** ha regresado, **A** ahora contiene una referencia en un OF que representa el resultado (que aun no esta disponible), de la llamada. El OF esta compuesto de un **Stub_V** y un **FutureProxy** como muestra la Figura 2.18.

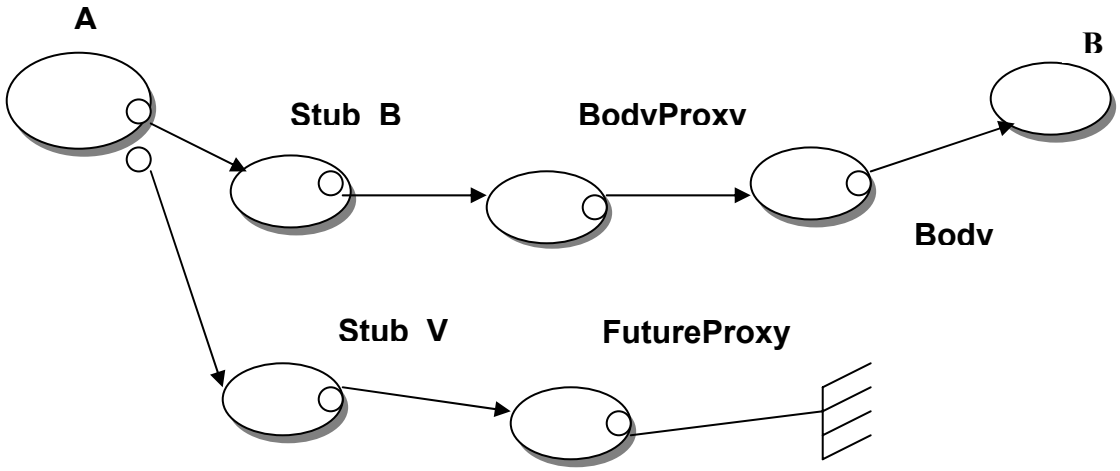

FIGURA 2.18 ESTADO DEL OBJETO FUTURO

Justo después de haber ejecutado **foo** en una instancia de **B**, el hilo del **Body** pone el resultado en el OF, que causa que en **FutureProxy** tenga una referencia dentro de **V** (ver Figura 2.19)**.**

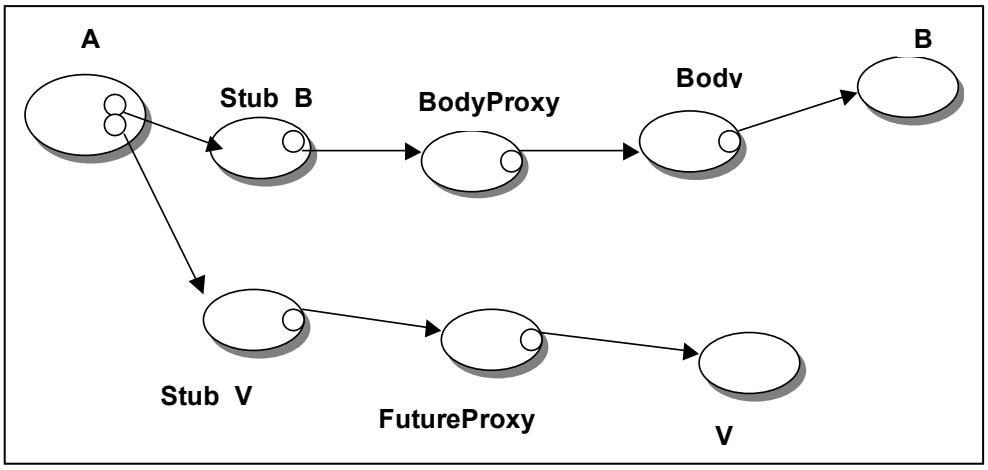

FIGURA 2.19 OBJETO FUTURO CON EL RESULTADO YA DISPONIBLE

Ahora se concentrará en como, cuando y cuales hilos de los diferentes métodos son llamados. Se tienen dos hilos: el que pertenece al subsistema **A** (primer hilo) y el hilo que pertenece al subsistema **B** (segundo hilo).

El primer hilo invoca **foo** en una instancia de **Stub_B**, el cual construye un objeto **MethodCall** y lo pasa a **BodyProxy** como parámetro de la llamada a **reify**. El proxy entonces checa el tipo que devuelve la llamada (en este caso **V**) y genera un objeto futuro de tipo **V** para representar el resultado de la invocación al método. El objeto futuro está realmente compuesto de un **Stub_V** y un **FutureProxy**. Una referencia dentro de este objeto futuro es puesta en el objeto **MethodCall**, el cual será útil una vez que la llamada es ejecutada. Ahora que el objeto **MethodCall** esta listo, este se pasa en una petición al **Body** del OA como parámetro. El cuerpo simplemente añade esta petición a la cola de peticiones pendientes y regresa inmediatamente. La llamada a **foo** que **A** emitió ahora regresa un OF de tipo **Stub_V**, que es una subclase de V.

En algún punto, posiblemente después de haber servido algunas otras peticiones, el segundo hilo (el hilo activo) recoge la petición emitida (por el primer hilo), entonces ejecuta una llamada incorporada en la invocación a **foo** en la instancia de **B** con los parámetros actuales almacenados en el objeto **MethodCall**. Como está especificado en su firma[6], ésta llamada regresa un objeto de tipo **V**. El segundo hilo es responsable de poner este objeto en un OF (esta es la razón del por que el objeto **MethodCall** contiene una referencia en el OF creado por el **FutureProxy**). La ejecución de esta llamada se termina y el segundo hilo puede seleccionar otra petición de la cola y ejecutarla.

Mientras tanto, el primer hilo continúa ejecutando el código del método invocado en la clase **A**. En algún punto, éste llama a **bar** en el objeto de tipo **Stub_V** que fue devuelto por la llamada a **foo**. Esta llamada es materializada gracias a **Stub_V** y procesada por el **FutureProxy**. Si el objeto que representa el futuro esta disponible (el segundo hilo se puso en el objeto futuro), la llamada se ejecuta sobre este y devuelve un valor al código invocado en **A**, lo cual se describe en la Figura 2.20.

[6] consiste en el nombre del método junto con el numero y tipo de sus argumentos

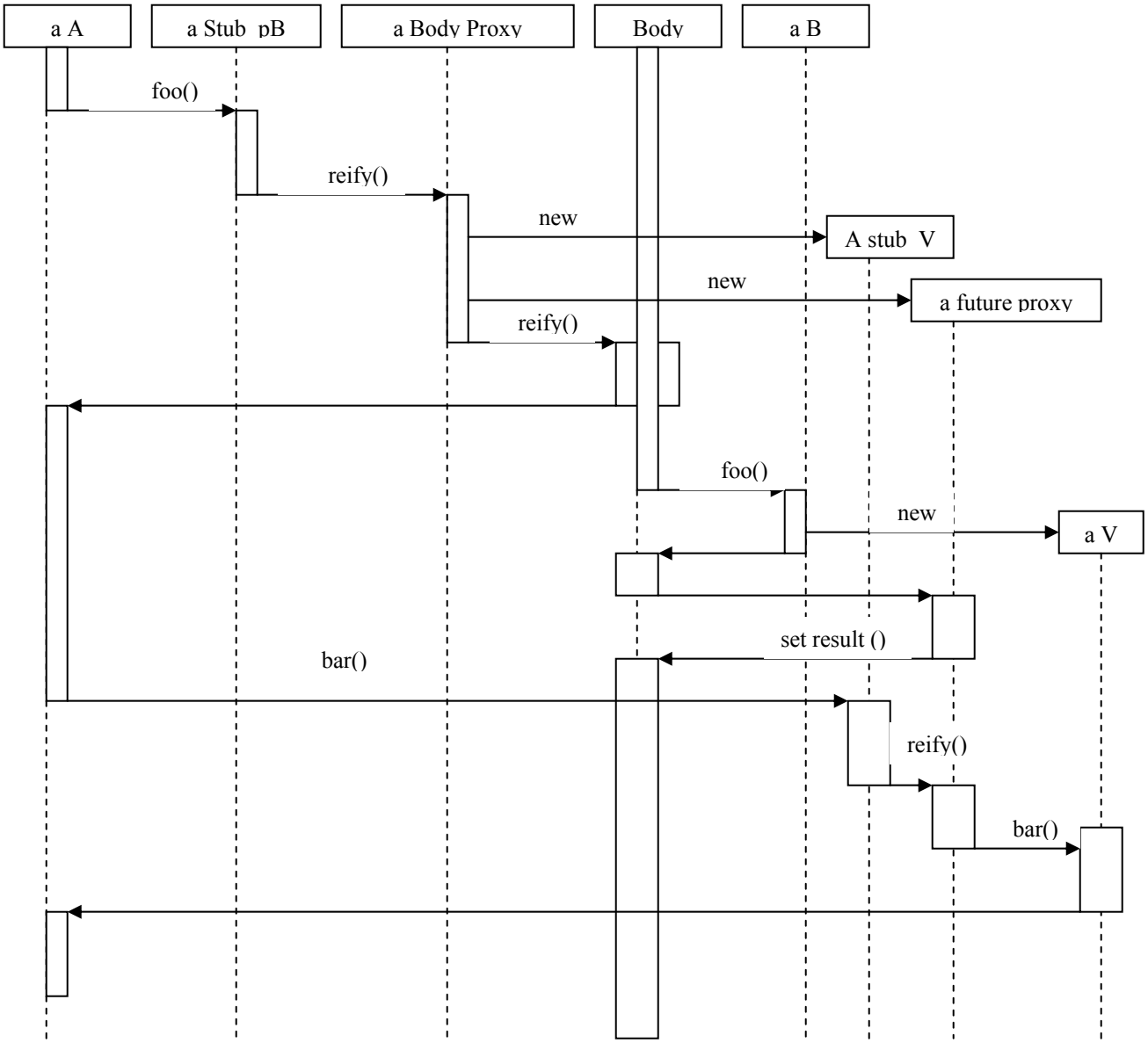

FIGURA 2.20 DIAGRAMA DE SECUENCIA DE LA LLAMADA ASÍNCRONA

Si el resultado no esta disponible aún, el primer hilo se suspende en **FutureProxy** hasta que el segundo hilo coloque el resultado en el OF tal como se muestra en el diagrama de secuencias de la Figura 2.21.

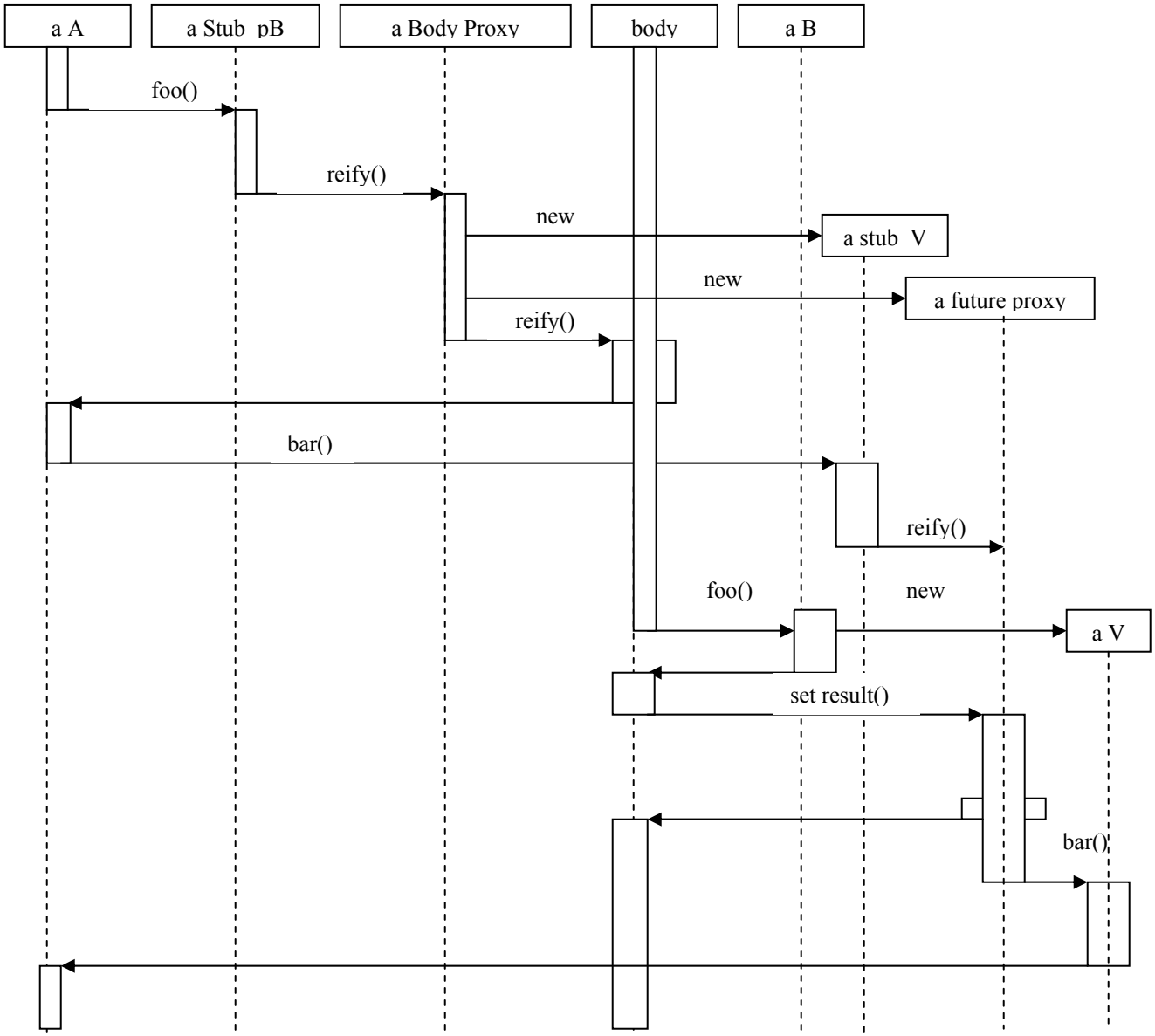

FIGURA 2.21 DIAGRAMA DE SECUENCIA DEL HILO SUSPENDIDO

2.11 Migración de objetos activos

2.11.1 Primitiva de migración

La migración de un OA puede ser provocado por el propio OA o por un agente externo. En ambos casos una primitiva se llama para aplicar la migración. Este método es **migrateTo()** que se accede desde un cuerpo migrable (un cuerpo que hereda de **AbstractMigratableBody**).

Para facilitar el uso de la migración, se proporcionan dos conjuntos de métodos estáticos en la clase ProActive.

El primer conjunto: Pone atención a la migración provocada por el OA que desea migrar. Los métodos confían el hecho de que el hilo que invoca, es el hilo activo del objeto activo.

- migrateTo(Object o): migran a la misma posición como un objeto activo existente.
- migrateTo(String nodeURL): migran a la posición dado por el URL del nodo
- migrateTo (Node node): migran a la posición del nodo dado.

El segundo conjunto: Pone atención a la migración provocada desde otro agente en vez del OA mismo. En este caso el agente externo debe tener una referencia al cuerpo del OA que quiere migrar.

- migrateTo(Body body, Object o, boolean priority): migran a la misma posición como un objeto activo existente.
- migrateTo(Body body, String nodeURL, boolean priority): migran a la posición dado por el URL del nodo
- migrateTo(Body body, Node node, boolean priority): migran a la posición del nodo dado.

2.11.2 Usando migración

Cualquier OA tiene la habilidad para migrar. Si éste referencia a algunos objetos pasivos, ellos también se migraran a la nueva ubicación. Dado que se confía en la serialización para el envío del objeto sobre la red, **el OA debe implementar la interfase Serializable.** Para migrar, un OA debe tener un método que contenga una llamada a la primitiva de migración. Esta llamada debe ser la última en el método, es decir, el método debe regresar inmediatamente. En la Figura 2.22 se muestra un ejemplo del uso del método en un OA:

```
public void moveTo(String t) {
    try {
        ProActive.migrateTo(t);
      }catch (Exception e) {
        e.printStack
    }
```

FIGURA 2.22 USO DEL MÉTODO MIGRATETO() EN UN OA

No se proporciona ninguna prueba para verificar si la llamada a **migrateTo()** es la última en el método, entonces si la regla no se cumple, esto puede generar un comportamiento inesperado. Ahora para hacer que este objeto se mueva, solo se debe llamar a su método **moveTo()**. Un ejemplo completo se encuentra en la Figura 2.23

```
import org.objectWeb.proactive.ProActive;
public class SimpleAgent implements Serializable {
    public SimpleAgent() {}
     public void moveTo(String t) {
       try {
            ProActive.migrateTo(t);
           }catch (Exception e) {
           e.printStackTrace();}
     }
}
public String whereAreYou() {
    try {
         return InetAddress.getLocalHost().getHostName();
        }catch (Exception e) {
         return "Búsqueda del local host fallida";}
}
public static void main (String[] args) {
  if (!args.length>0)) {
     System.out.println("Uso: java migration.test.TestSimple
                                    hostname/NodeName");
            System.exit(-1);}
          SimpleAgent t = null;
          try { //crea un SimpleAgent en esta MVJ
               t= (SimpleAgent)
                        ProActive.newActive("migration.test.SimpleAgent",null);
              }catch (Exception e) {
               e.printStackTrace();}
         //migran el SimpleAgent a la posición identificada por el nodo URL
         //se asume que el ya nodo existe
         t.moveTo(args[10]);
        System.out.println(El objeto activo esta ahora en el host " + t.whereAreYou());
       }
}
```

FIGURA 2.23 EJEMPLO COMPLETO USANDO MIGRACIÓN DE OBJETOS

La clase SimpleAgent implementa **Serializable,** de tal manera que los objetos creados serán capaces de migrar. Se necesita proporcionar un constructor vacío para evitar efectos colaterales durante la creación de OA. Este objeto tiene dos métodos, **moveTo()** el cual lo migra a la posición especificada y, **whereAreYou()** el cual regresa el nombre del anfitrión de la nueva ubicación del objeto.

En el método principal (main), primero es necesario crear un OA, el cual se hace a través de la llamada a **newActive()**. Una vez que se ha hecho esto, se llaman a los métodos en éste como en cualquier objeto. Se invoca su método **moveT** se migra al nodo especificado como parámetro y entonces pregunta cuál es su posición actual.

2.11.3 Manejo de atributos no-serializables

La migración de un objeto activo usa la serialización. Desafortunadamente, no todos los objetos en el lenguaje Java son serializables. Existe un método simple que trata con tales atributos, en el caso de que sus valores no necesiten ser conservados. Para casos más complejos, el lector puede ver las especificaciones de Java RMI.

Cuando una excepción **NotSerializable** se lanza, el primer paso para resolver el problema es identificar la variable responsable, es decir la que no es serializable, entonces antes de la declaración de la variable, poner la palabra reservada **transient,** esto indica que el valor de esta variable no debería ser serializable. Después de la primera migración este campo es cambiado a nulo ya que el valor no tiene que ser guardado. La variable se tiene que reconstruir bajo el arribo del OA en su nueva ubicación. Esto se puede hacer fácilmente proporcionando un método estándar en el OA.

```
private void readObject(java.io.ObjectInputStream in) throws
                    java.io.IOException, ClassNotFoundException;
```

Consultar la interfase **Serializable** en el JavaDoc estándar.

2.12 Protocolo Metaobjeto: MOP

2.12.1 Implementación: un protocolo metaobjeto

ProActive está construido en la cima del protocolo metaobjeto (MOP), que permite la abstracción de la invocación a los métodos y de las llamadas al constructor. El MOP no esta limitado a la implementación de la biblioteca de objetos remotos transparentes, si no que éste además proporciona un ambiente de ambiente de trabajo abierto para implementar poderosas bibliotecas para el lenguaje Java.

Como cualquier elemento de ProActive, el MOP está totalmente escrito en Java y no requiere modificación alguna o extensión de la máquina virtual de Java, de manera opuesta a otros protocolos metaobjeto para Java. Éste hace extensivo el uso de la API Java Reflection, el cual requiere de JDK 1.1 o mayor. Se requiere JDK1.2 para suprimir las verificaciones de control de acceso por defecto del lenguaje Java cuando se ejecuta un método no público materializado o llamadas al constructor.

2.12.2 Principios

Si el programador desea implementar un nuevo metacomportamiento usando el MOP, debe de escribir una clase concreta (como opuesto a lo abstracto) y una interfase. La clase concreta proporciona una implementación para el metacomportamiento que se desea obtener mientras que la interfase contiene su parte declarativa.

La clase concreta implementa la interfase **Proxy** y proporciona una implementación para el comportamiento dado a través del método **reify:**

puclic Object reify (MethodCall c) throws Throwable;

Este método toma una llamada materializada como un parámetro y regresa el valor obtenido de la ejecución de esta llamada materializada. Se proporciona un empaquetamiento y desempaquetamiento automático de tipos de primitivas. Si la ejecución de la llamada termina abruptamente por el lanzamiento de una excepción, ésta es propagada al método invocador, tal y como si la llamada no hubiese sido materializada.

La interfase que contiene la parte declarativa del metacomportamiento tiene que ser una subinterface de **Reflect** (la interfase raíz de todos los metacomportamientos implementados usando ProActive). El propósito de esta interfase es declarar el nombre de la clase proxy que implementa el comportamiento dado. Entonces cualquier instancia de una clase que tenga implementada esta interfase será creada automáticamente con un proxy que implementa este comportamiento, considerando que esta instancia no es creada usando la palabra clave **new** sino a través de un método estático: **MOP.newInstance**. Esta es la única modificación que se requiere para el código de la aplicación. Otro método estático **MOP.newWrapper**, agrega un proxy a un objeto ya existente; la función de ProActive **turnActive**, por ejemplo, esta implementada a través de esta característica.

Ejemplo de un comportamiento diferente: EchoProxy

Se muestra en la Figura 2.24 la implementación de un metacomportamiento simple pero útil: para cada llamada materializada, el nombre del método invocado es impreso en el flujo de salida estándar y la llamada es entonces ejecutada. Esto puede ser un punto de inicio para la construcción de ambientes de depuración o de perfiles.

```
class EchoProxy extends Object implements Proxy {
// Aquí el constructor y las variables de declaración
// {...}
public Object reify (MethodCall c) throws Throwable {
     System.out.println (c.getMethodName());
     Return c.execute (targetObject);
  }
}

interfase Echo extends Reflect {
    public String PROXY_CLASS= "EchoProxy";
}
interfase Echo extends Reflect {
public String PROXY_CLASS= "EchoProxy";
}
```

FIGURA 2.24 IMPLEMENTACIÓN DE UN METACOMPORTAMIENTO

2.12.3 Instanciación con el metacomportamiento

Instanciar un objeto de cualquier clase con este metacomportamiento se puede hacer de tres formas diferentes:

1. basado en instanciación
2. basado en clase
3. basado en objeto

Suponga que se desea instanciar un objeto **Vector** con un comportamiento ***Echo***:

- Con código Java estándar debería ser:
  Vector v = new Vector(3);

- Con código ProActive, la declaración del metacomportamiento basado en instanciación (él ultimo parámetro es nulo por que no se tiene otro parámetro adicional para pasar al proxy) es:
  Object[] params = {new Integer (3) };
  Vector v = (Vector) MOP.newInstance("Vector", params, "EchoProxy", null);

- Con declaración basada en clase
  public class MyVector extends Vector implements Echo {}
  Object[] params = {new Integer (3)};
  Vector v = (Vector) MOP.newInstance("Vector", params, null);

- Con declaración basado en objeto
  Vector v = new Vector (3);
  v=(Vector) MOP.newWraper("EchoProxy",v);

Este el única manera de proporcionar un metacomportamiento a un objeto que se crea en un lugar donde no se puede editar el código fuente. Un ejemplo típico puede ser un objeto devuelto por un método que es parte de una API distribuida como un archivo JAR, sin el código fuente. Note que cuando se usa **newWrapper**, la invocación del constructor de la clase Vector no se abstrae.

2.12.4 La interfase Reflect

Todas las interfases usadas para declarar *metacomportamientos* heredan directa o indirectamente de **Reflect.** Esto da lugar a una jerarquía de metacomportamientos tal como se muestra en la Figura 2.25:

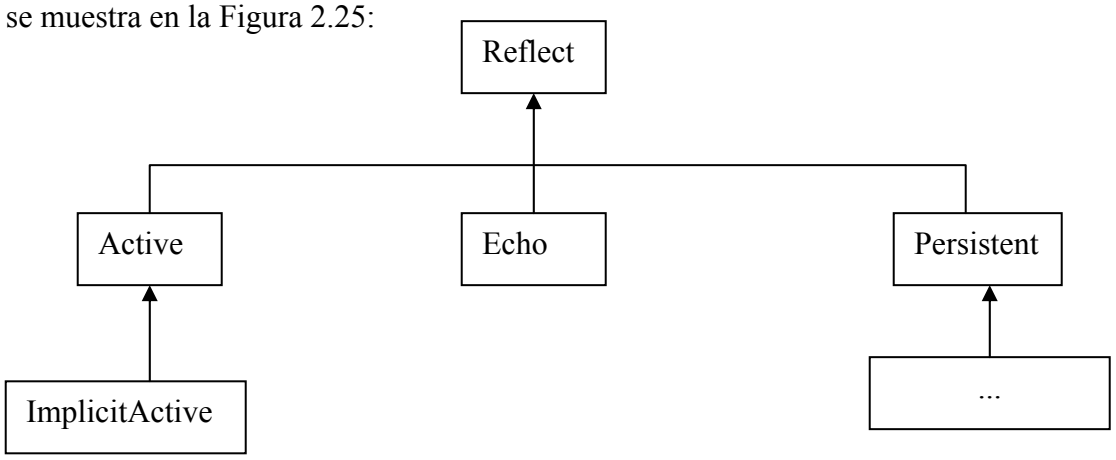

FIGURA 2.25 DIAGRAMA DE INTERFASE Y SUBINTERFASES DE REFLECT

Note que **ImplicitActive** hereda de **Active** para resaltar el hecho de que la sincronización implícita siempre confía en algún mecanismo explicito oculto. Las interfaces heredan de **Reflect**, pueden entonces ser lógicamente agrupadas y ensambladas usando herencia múltiple para construir nuevos metacomportamientos de los ya existentes.

2.12.5 Limitaciones

Debido a su compromiso de ser 100 % biblioteca de Java, el MOP tiene algunas limitaciones

- Las llamadas enviadas a instancias de clases final (las cuales incluyen todos los arreglos) no pueden ser materializadas.
- Los tipos de primitivas no pueden ser materializada por que ellos no son instancias de clases estándar
- Las clases Final no pueden ser materializada por que no pueden ser subclasificadas.

2.13 Conclusiones

La biblioteca ProActive, está en constante crecimiento con la finalidad de mejorar su capacidad de procesamiento, convirtiéndose ya en una alternativa real para el desarrollo de aplicaciones distribuidas.

Con sus características esenciales como el objeto activo migrable, su protocolo MOP, la reutilización de código, la programación transparente aunado al auge del nuevo paradigma de agentes en el campo de la computación y mas concretamente los agentes móviles que son la evolución consecuente del concepto de objeto de la tecnología orientada a objetos. Se vuelve imperante una infraestructura genérica para el manejo de agentes, la cual siente las bases para futuros desarrollos de aplicaciones distribuidas con agentes.

Capítulo 3 Agentes móviles

3.1 Definición de agente

La palabra "Agente" (del latín agens) se aplica a una sustancia activa, una persona o cosa que produce un efecto, persona que obra con poder de otra, persona que tiene a su cargo una agencia para gestionar asuntos ajenos o prestar determinados servicios, un representante según el diccionario de la real academia española. A pesar de que este concepto ha llamado la atención de investigadores en las áreas de interfaz Hombre-Máquina, Sistemas Distribuidos e Inteligencia Artificial, aun no se ha llegado a una definición generalmente aceptada de lo que es "Agente".

En la actualidad se usan diversas términos para describir otros comportamientos de los agentes: agentes inteligentes, interfases inteligentes, interfases adoptivas, knowhots (conocimiento basado en robots), softbots (software para robots), taskbots (tareas basada en robots), agentes personales, agentes autónomos, agentes de red, solo por mencionar algunos.

Sánchez define un agente como "*una entidad autónoma o semiautónoma que cumple con una misión bien definida*" en [SAN96]. Mientras que para Wooldridge, distingue que el termino de agente es usado para referirse a una unidad de hardware (por lo general) o software que tiene las siguientes propiedades:

Autonomía, los agentes operan sin la intervención directa del ser humano y tiene un tipo de control sobre sus acciones y sus estado interno.

Sociable, los agentes interactúan con otros agentes (y posiblemente con humanos) vía un lenguaje de comunicación de agentes.

Reactividad, los agentes perciben su ambiente (el cual puede ser el mundo físico, un usuario vía una interfaz gráfica, una colección de agentes, la Internet o todo lo anterior) y responde a los cambios que se dan en el ambiente.

Y le atribuyen otros conceptos tales como:
Movilidad, la cual es una habilidad para moverse a través de la red.

Veracidad, se asume que un agente no puede comunicar información falsa.

Benevolencia, asume que el gente no tiene conflictos de metas y que cada agente intentará responder para lo cual fue requerido.

Racionalidad, asume un agente esta orientado a metas, [WOO95].

Y una de las definiciones formales de agente es dada por Lecky Thompson en [LEC01]: *"Un agente es una pieza de software que ejecuta una tarea dada usando información obtenida de su entorno para actuar de forma apropiada para completar la tarea con éxito. El software debe ser capaz de adaptarse a sí mismo en base a los cambios que ocurren en su entorno, para que un cambio en circunstancias le permita aún obtener el resultado deseado*".

3.2 Taxonomía de agentes

La propuesta Sánchez en [SAN97] de una taxonomía de los agentes engloba y reconcilia a las diferentes definiciones y puntos de vista, además de que proporciona las bases para taxonomías más detalladas (ver Figura 3.1).

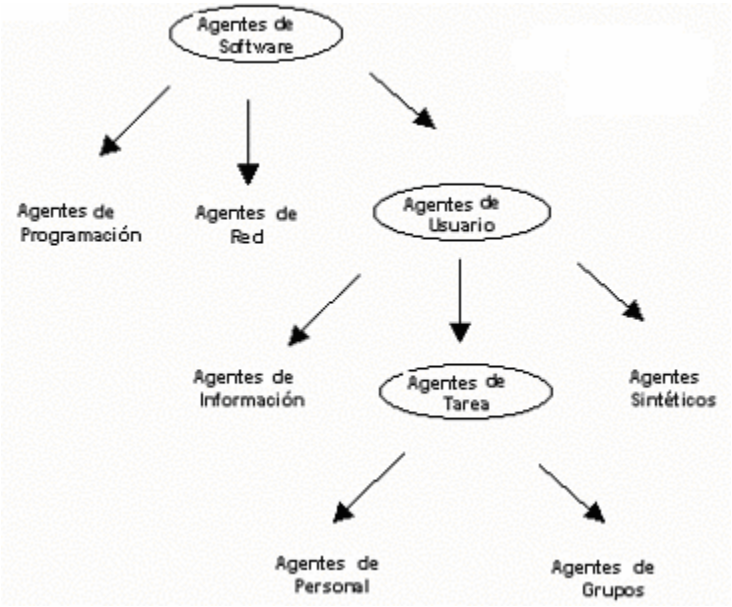

FIGURA 3.1 TAXONOMÍA DE AGENTES

Esta propuesta plantea tres amplias clases de agentes:
1.- De programación, los cuales son mecanismos de abstracción usados por desarrolladores de software para visualizar, diseñar e implementar sistemas complejos.
2.-De red, que son entidades que emigran autónomamente entre los nodos de una red en un ambiente distribuido para realizar una tarea encomendada por el usuario y permiten que los usuarios finales interactúen con los sistemas programador.
3.-De usuario, que son abstracciones de cómputo permitiendo delegar tareas bien definidas.

3.3 Concepto de agente móvil

Los siguientes términos, agentes de red, agentes transportables, agentes itinerantes, generalmente son usados para referirse al mismo concepto – *agente móvil*. La característica distintiva de este agente es su capacidad de migrar.

Los términos anteriores dan pie a diversas definiciones:
Un agente transportable es un programa que puede migrar de una máquina a otra en una red heterogénea y esta dado en [GRA95]. Este tipo de agente debe ser portable a través de plataformas, debe ser capaz de elegir cuando y donde transportarse a sí mismo, debe ser capaz de duplicarse a sí mismo y debe poder comunicarse con otros agentes para intercambiar información.

Los agentes móviles son procesos computacionales de software capaces de moverse en redes de área amplia (WAN) tales como el WWW, interactuando con diferentes anfitriones, recolectando información en nombre de su dueño y regresando a casa después de ejecutar las tareas delegadas por el usuario, este concepto lo manejan en [NEW96]. Esas tareas pueden ir desde una reservación de vuelo hasta el manejo de una red de telecomunicaciones. Los agentes móviles son autónomos y cooperan. Por ejemplo, pueden cooperar o comunicarse con otros agentes intercambiando datos o información.
Los agentes transportables son aquellos que soportan el movimiento de cómputo cliente a un sitio donde se encuentra un recurso remoto, esta definición esta dada en [KOK94]. Son capaces de suspender su ejecución, transportarse ellos mismos a otros anfitriones en la red y reanudar la ejecución desde el punto en el cual fueron suspendidos. Consumen pocos recursos de red y pueden soportar sistemas que no tienen una permanente conexión a la red, tales como computadoras móviles.

Los agentes móviles son objetos que consisten de código, datos de ejecución que puede ir más allá de dominios protegidos este concepto se encuentra en [KTM97].

Un agente móvil es un conjunto de objetos ejecutando un cálculo en nombre de un usuario. Este cálculo es ejecutado en una plataforma de ejecución de agentes que controla la ejecución del agente esta referenciado en [VIT96]. Un agente puede requerir moverse, causando que su cálculo sea interrumpido y reanudado en otro lugar.

Y por último según en [WHI96] un agente móvil es un programa:

-Que una persona u organización enviste con su autoridad
-Que puede correr sin supervisión por un largo periodo de tiempo.
-Que puede conocer e interactuar con otros agentes
-Que puede ejecutarse en diferentes sistemas de cómputo y en diferentes etapas de su vida.

Un intento más técnico orientado a la tecnología de objetos y agentes móviles es el dado en [HBS96]: *"Un agente móvil es un objeto especial que tiene un estado de datos (otros objetos no agentes, estructuras y bases de datos), un estado de código (las clases del agente y otras referencias a objetos) y un estado de ejecución (el control de procesos que se ejecutan en el agente*)."

Como ninguna de estas definiciones resulta completa, en vez de dar una definición formal, se suele proporcionar la lista de características que se espera que un agente deba tener para poder tener una idea de lo que un agente puede ser. Las características siguientes suelen tenerlas los agentes móviles que serán el tipo de agentes a los que está dedicado este apartado.

- Es autónomo o semiautónomo de manera que él decide cómo, cuándo y a dónde migrar.
- Esta orientado a ejecutar tareas, a veces en nombre del usuario y otras basándose en los cambios de su ambiente
- Se envía como objeto, a través de plataformas conservando además de su código, los datos y su estado de ejecución.
- Es asíncrono, debido a que tienen su propio proceso o hilo de ejecución. Por lo tanto el agente se ejecuta asincrónicamente respecto a los otros procesos que se estén ejecutando en el nodo.
- Es capaz de comunicarse con su dueño, con otros agentes y con el medio.
- Puede operar sin conexión, es decir, que puede ejecutar sus tareas aun cuando la conexión a red no este funcionando; si el agente necesita trasladarse y la red no está activa, el agente puede esperar o desactivarse hasta que la conexión se restablezca.
- Puede suspender su ejecución, transportarse a otro anfitrión y reanudar su ejecución desde el punto en el cual se suspendió.
- Es capaz de duplicarse
- Puede reaccionar a cambio en su ambiente, modificando su conducta debido a las acciones generadas por otros agentes, debido a su experiencia propia o por la intervención directa del programador o usuario.

Aunque aún no hay un solo agente que posea todas estas habilidades, existen sistemas de agentes prototipo que poseen muchas de ellas. Además proporcionan una buena idea de lo que un agente puede hacer, así como las características que deben evaluarse al examinar la calidad de un sistema de agentes en específico.

3.4 Infraestructura de agentes móviles genérica

Para ser útil un agente necesita interactuar con su nodo y otros agentes, debe acceder información que la máquina ofrece y/o negociar con otros agentes sobre el intercambio de servicios. Los agentes deben ser capaces de moverse dentro de redes heterogéneas de computadoras. Esto es posible solamente si existe un marco de trabajo común para operaciones de agentes a través de la red completa: una infraestructura de agentes estandarizada.

Esta infraestructura debe ofrecer [GRAY95]:
-Soporte básico para la movilidad y comunicación de agentes
-Proteger a la computadora de accesos no autorizados
-Salvaguardar la integridad de los agentes, tanto como sea posible

En la literatura de agentes móviles se encuentran las siguientes infraestructuras propuestas, por diversos autores.

3.4.1 Infraestructura Crystaliz

Crystaliz, Inc., sugiere un modelo común [CRY97] que extrae las características más generales de los sistemas de agentes móviles que existen con el fin de estandarizar y promover interoperabilidad entre dichos sistemas (ver Figura 3.2).

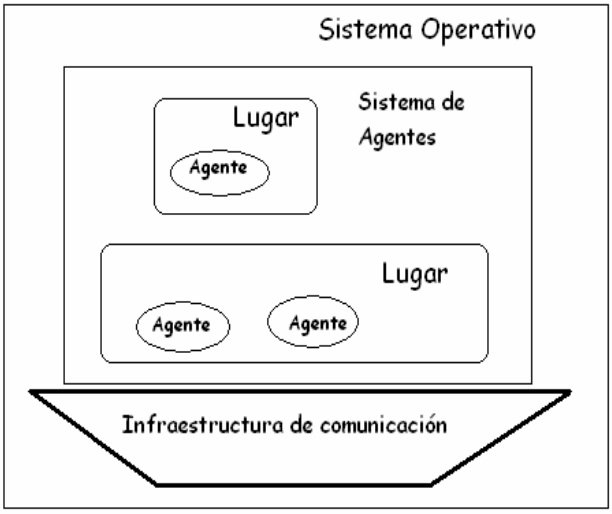

FIGURA 3.2 INFRAESTRUCTURA CRYSTALIZ

- El sistema de agentes es una plataforma que puede crear, interpretar, ejecutar, transferir y terminar agentes.
- Un lugar es donde el agente reside, es un contexto en un sistema de agentes en el cual un agente puede ejecutarse.
- Toda la comunicación entre sistemas de agentes se hace a través de la infraestructura de comunicación (IC).

Se identifican tres tipos de interacción de agente, en la infraestructura:

- Creación de agentes remotos. Un programa cliente interactúa con el sistema de agente, pidiéndole crear un agente de una clase particular.
- Transferencia de agentes. Cuando un agente decide transferirse a otro sistema de agente, el sistema de agentes le crea una solicitud de viaje.
- Invocación de métodos de agentes. Un agente puede invocar el método de otro agente u objeto, si está autorizado para hacerlo y tiene una referencia al objeto.

Funciones de un sistema de agentes:

1. Transferir un agente, lo cual puede incluir iniciar una infraestructura de agente, recibir un agente, y transferir clases.
2. Crear un agente,
3. Proporcionar nombres únicos a los agentes,

4. Soportar el concepto de región (conjunto de sistemas de agentes)
5. Hallar un agente móvil
6. Asegurar un ambiente seguro para operaciones de agentes.

3.4.2 Infraestructura según Lingnau

Lingnau y colegas, proponen en [LDD95] una infraestructura para agentes móviles basada en el protocolo de transferencia de hipertexto (HTPP) el cual proporciona movimiento al agente a través de redes heterogéneas así como comunicación entre éstos. Soporta agentes escritos en diversos lenguajes y permite implementar una variedad de esquemas de interacción basados en un mecanismo general para comunicación de agentes. La base de esta infraestructura es la idea del servidor de agentes (ver Figura 3.3).

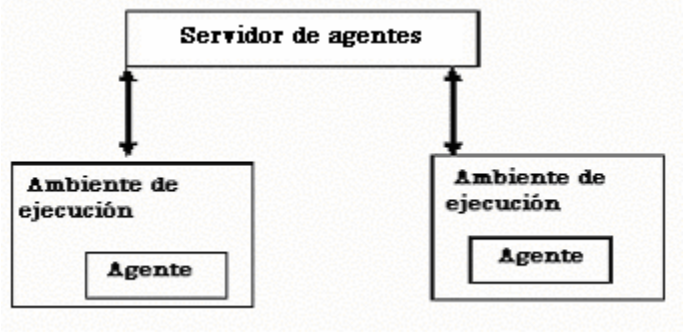

FIGURA 3.3 INFRAESTRUCTURA LINGNAU

- El servidor de agentes es un programa que corre en cada computadora, que es accesible a los agentes y está a cargo de que los agentes corran en esa computadora; sus tareas son:
  - Aceptar agentes, crear ambientes de ejecución apropiados, ejecución de los agentes y terminarlos.
  - Organizar la transferencia entre otros nodos
  - Manejar la comunicación entre agentes, así como entre los agentes y sus dueños y hacer autenticación y control de acceso para todas las operaciones del agente.
  - Participación en el manejo de operaciones de red.

Cada servidor de agentes conoce sobre otros servidores de agentes en su vecindad y esta información la hace disponible para los agentes, quienes la usan para elegir un nuevo destino, cuando ellos deciden dejar la máquina.

- El ambiente de ejecución existe en cada servidor con agentes ejecutándose, este funciona como interfaz entre el agente y su nodo, permitiendo que los recursos de la máquina estén disponibles para el agente de una manera controlada.

3.4.3 Infraestructura según Stone

Stone y colegas proponen en [SZB96] una infraestructura para agentes móviles cuyos componentes principales son (ver Figura 3.4):

- Agentes móviles
- Lenguaje de agentes
- Lugares de reunión de agentes (AMP,"*agente meeting places*")
- Una máquina

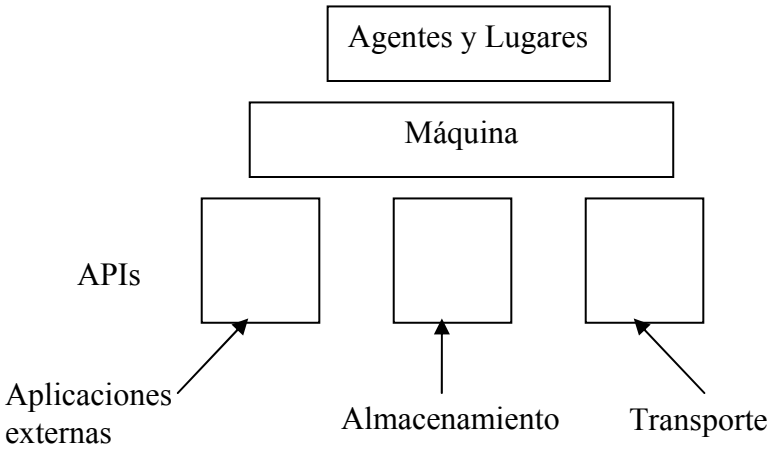

FIGURA 3.4 INFRAESTRUCTURA STONE

El agente puede estar escrito en varios lenguajes de programación y puede transportar conocimiento expresado en varias formas. Debe ser capaz de entablar un diálogo con el lugar de reunión de agente hasta que se ejecute o se rechace. Puede ejecutarse hasta su terminación o puede elegir suspender su actividad y moverse a otro lugar de reunión de agentes y continuar su ejecución ahí.

Una máquina es un programa residente en el servidor que implementa un marco de trabajo para el agente, manteniendo y ejecutando los AMP que contiene, así como los agentes que ocupan los AMP. En general, la máquina es un intérprete para el lenguaje usado para implementar el ambiente de trabajo del agente.

La máquina se comunica con el nodo a través de tres aplicaciones API. Los API son usados para manejar almacenamiento, transportar agentes y comunicarse con las aplicaciones externas.

3.4.4.-TACOMA, una infraestructura para agentes móviles

Esta infraestructura se compone de:

- Agentes, cuando migran se ejecutan desde el comienzo del programa en vez de continuar después del punto de migración.
- Carpetas, las usan los agentes para comunicarse.
- Portafolios, es la colección de carpetas asociados con un agente.
- Gabinetes, es la colección de carpetas estacionarias necesarias para propósitos de almacenamiento de datos permanentes.

Los agentes TACOMA desarrollados en la Universidad de Troms y la Universidad de Cornell, están escritos en Tcl/Horus (versión de Tcl, Tool Command Language) que

proporciona comunicación y tolerancia a fallas descritos en [JRS95]. La abstracción más importante en TACOMA es la operación Met, la cual se usa para que un agente ejecute otro agente. Este es un concepto vital usado para la comunicación y sincronización entre agentes. La operación Met debe tener un punto de entrada, en el sitio destino.

3.5 Requerimientos para el desarrollo de agentes móviles

Las cuatro arquitecturas presentadas, tienen en común el hecho de que para poder ejecutar un agente, se necesita de un ambiente adecuado y de un sistema o servidor de agentes que controle el acceso al ambiente. Dicho sistema debe ser capaz de monitorear las instrucciones a ser ejecutadas por los agentes y debe proveer las facilidades para migración y recepción de agentes, así como mecanismos para que los agentes se conozcan e intercambien datos.

La implementación de agentes móviles implica cubrir algunos requerimientos entre los que se pueden mencionar [SAH97]:

- Un lenguaje portable. El código de un agente móvil debe ser aceptado en diferentes plataformas sin requerir una recopilación previa.
- Un lenguaje intérprete. Se requiere de un lenguaje intérprete muy útil para la portabilidad en el marco de una máquina virtual. La característica de intérprete es también importante cuando el servidor, desea verificar el código antes de ejecutarlo. Los lenguajes intérpretes permiten que la ejecución sea monitoreada y algunas acciones sean abortadas cuando infringen la política de seguridad.
- Un ligado tardío, dada la naturaleza de los agentes, el lenguaje tiene que ser tan flexible como sea posible. Si el agente no conoce a priori a que tipo de servidor esta migrando, es necesario un enlace dinámico. Esto es adecuado cuando la información completa acerca de la naturaleza de los objetos solamente es conocida en tiempo de ejecución.
- Una representación común de conocimiento. Los agentes son autónomos en el sentido de que ellos acarrean todo lo que necesitan para vivir como programas estándar, esto incluye código y datos. Así que es importante tener una manera estándar de representar datos y conocimientos si se desean involucrar agentes en un ambiente de trabajo de colaboración.
- Una abstracción del ambiente remoto. Cuando se programa el comportamiento de un lenguaje el usuario debe tener una representación abstracta de los recursos remotos que el agente será capaz de usar. Esto significa que debe existir una representación común de servidores remotos.
- Programas adaptables. Los agentes deben ser capaces de enfrentarse con cualquier tipo de ambiente. La solución más sencilla para un agente seria moverse a otra computadora si las capacidades ofrecidas localmente no son autosuficientes. Sin embargo, los agentes deben ser capaces de manejar algunas de las restricciones locales como: almacenamiento limitado de disco, limitación de memoria y limitación de tiempo de CPU.

- Compartir recursos. Una de las ideas detrás del paradigma de agentes es tener algunos servidores compartidos que proporcionen recursos para los agentes y por consiguiente sean capaces de compartir dichos recursos de una manera óptima.
- Un modelo de negocios. En este sentido hay una serie de cuestiones a estudiar por ejemplo: ¿Cómo definir el precio?, que tipo de precio manejar (pago por ver, suscripción, etc.), que tarifas imponer (modular dependiendo de la carga del servidor o de la hora de acceso, cuotas especiales para grupos, etc.)

Resulta oportuno mencionar que hoy en día los sistemas de agentes móviles más populares son los basados en Java. Estos sistemas son capaces de transportar el estado de datos, es decir sus bases de datos y estructuras; de igual forma pueden transportar su estado de código, como lo son las clases y objetos que requiere, pero no su estado de ejecución, que se refiere al punto exacto que se estaba ejecutando al detener la ejecución del agente. Esto es debido a que la máquina virtual de Java no permite acceder a estos recursos.

3.6 Tecnologías de agentes móviles

La primera implementación comercial del concepto de agentes móviles fue la tecnología de Telescript de General Magic que intentó permitir acceso automático e interactivo a las redes de computadoras usando agentes móviles [TAV96].

Telescript fundó las bases de la tecnología de agentes móviles y es reconocido precursor de la misma. Estos conceptos fueron tan bien diseñados que han sido utilizados por los nuevos sistemas de agentes de nuestros días.

La tecnología de agentes modela una red de computadoras, tan grande como una colección de lugares que ofrecen un servicio a los agentes móviles que entran en él. Así pues, un lugar es un sitio de interacción de agentes que les ofrece un servicio y les facilita la continuación de sus tareas a todos aquellos agentes a los que les haya sido permitido entrar. En la tecnología de Telescript un lugar era ocupado permanentemente por un agente distinguido. Este agente estacionario representaba al lugar y proveía su servicio.

3.7 Particularidades de los agentes móviles para la construcción de sistemas distribuidos.

Los agentes móviles proporcionan un enfoque claramente diferenciado del tradicional para la construcción de sistemas distribuidos. A continuación se describen aquellos aspectos claramente particulares de este nuevo enfoque:

Los agentes se ejecutan de forma asíncrona y autónoma: frecuentemente los dispositivos móviles tienen que depender de conexiones de redes caras o frágiles.

Esto es, tareas que requieren una conexión abierta continuamente entre un dispositivo móvil y una red no serán en la mayoría de los casos económica o técnicamente factible. Las tareas pueden ser incrustadas en agentes móviles, que serán despachados hacia la red. Después de ser enviados, los agentes móviles llegan a ser independientes de la creación de

procesos y pueden operar de forma asíncrona y autónoma. El dispositivo móvil puede conectarse nuevamente más tarde para consultar al agente el resultado de la tarea.

Los agentes móviles se adaptan dinámicamente y tienen la habilidad de percibir su entorno y reaccionar de forma autónoma a cambios. Un conjunto de agentes móviles posee la habilidad única de distribuirse a sí mismos entre los servidores en una red de tal forma que mantienen la configuración óptima para la solución de un problema particular.

Son heterogéneos por naturaleza. La computación en redes es fundamentalmente heterogénea, con frecuencia desde las perspectivas de hardware y software.

Como los agentes móviles son generalmente independientes de la computadora y de la capa de transporte y dependientes sólo de su entorno de ejecución, proporcionan las condiciones óptimas para una integración transparente.

Son robustos y tolerantes a fallos. La habilidad de los agentes móviles para reaccionar dinámicamente a situaciones desfavorables y eventos hace más fácil construir sistemas distribuidos robustos y tolerantes a fallos. Si un servidor está siendo apagado, todos los agentes que se ejecutan en ese servidor serán avisados y se les dará tiempo suficiente para que se envíen a sí mismos y continúen sus operaciones en otro servidor de la red.

3.8 Aplicaciones de los agentes móviles

Los agentes móviles pueden ser utilizados para desarrollar una gran cantidad de aplicaciones. Estas aplicaciones pueden estar basadas en la tecnología de agentes, o bien pueden ser complementos de aplicaciones basadas en las tecnologías de orientación a objetos. Es decir, puede haber un sistema convencional para realizar búsquedas remotas utilice la tecnología de agentes móviles.

De una u otra forma las aplicaciones más importantes que se pueden llevar a cabo con agentes móviles están entre las siguientes [VEN97]:

Recolección de datos de distintos sitios. Una de las mayores diferencias entre el código móvil, como los *applets* de Java, y los agentes móviles es el itinerario, mientras que el código móvil usualmente viaja sólo de un punto a otro, los agentes móviles tienen un itinerario y pueden viajar secuencialmente a muchos servidores. De ahí que una aplicación natural de los agentes móviles sea la recolección de información a través de muchas computadoras enlazadas a una red. Un ejemplo de esta clase de aplicación es una herramienta de copias de seguridad que periódicamente debe supervisar cada disco instalado en cada computadora que se encuentra enlazado a la red. Aquí, un agente móvil podría navegar la red, recolectar la información acerca del estado de la copia de seguridad de cada disco y entonces regresar a su posición de origen y hacer un informe.

Búsqueda y filtrado. Dado el constante incremento en la cantidad de información disponible en Internet y en otras redes, la actividad de recolectar información de una red implica la búsqueda entre grandes cantidades datos de unas cuantas piezas relevantes de información. Eliminar la información irrelevante puede ser un proceso que consume mucho

tiempo. En nombre de un usuario, un agente móvil puede visitar muchos servidores, buscar a través de la información disponible en cada servidor, y construir un índice de enlaces a las piezas de información que concuerdan con el criterio de búsqueda. El filtrado y la búsqueda muestran un atributo común a muchas aplicaciones potenciales de agentes móviles: el conocimiento de las preferencias del usuario.

Monitorización. En algunas ocasiones la información no está distribuida a través de un espacio como puede ser un conjunto de computadoras, sino a través del tiempo. Nueva información constantemente está siendo generada y publicada en la red. Los agentes pueden ser enviados para esperar por ciertas clases de información hasta que ésta sea generada o se encuentre disponible. Por ejemplo un agente personalizado para la reunión de noticias. Un agente podría monitorizar varias fuentes de noticias para determinado tipo de información de interés para su usuario, e informarle cuando alguna información relevante esté disponible.
Comercio electrónico. Es una aplicación apropiada para el uso de la tecnología de agentes móviles. Un agente móvil podría realizar compras, incluyendo la realización de órdenes de compra y potencialmente pagar. Por ejemplo, si se quiere volar de un sitio a otro, un agente podría visitar las bases datos de los horarios de vuelo y los precios de varias líneas aéreas, encontrar el mejor precio y horario de salida, hacer la reserva e incluso pagar con un número de tarjeta de crédito.

Supercomputadora virtual (*cálculos en multiprocesos*). Los cálculos complejos con frecuencia pueden ser descompuestos en unidades discretas para la distribución en una pila de servidores o procesos. Cada una de estas unidades discretas puede ser asignada a un agente, el cual es entonces enviado a un sitio remoto en donde el trabajo es realizado. Una vez terminado, cada agente puede regresar a casa con los resultados, que pueden ser agregados y sumados.

3.9 Ventajas y desventajas

3.9.1 Ventajas

- Reduce costos de comunicación, podría haber una gran cantidad de información que necesita ser examinada para determinar su relevancia. Transferir esta información puede consumir tiempo y atascar la red. Imagínese tener que transferir muchas imágenes solas para elegir finalmente una. Es mucho mas natural tener un agente que "vaya" a esa localidad, haga una búsqueda / elección y solamente transfiera la imagen elegida de regreso a través de la red. Esto evita la necesidad de hacer conexiones de red costosas entre computadoras remotas tan requeridas en llamadas de procedimientos remotos (RPC). Esto proporciona una alternativa mucho mas barata en ancho de banda y en tiempo de acceso.
- No se limita a recursos locales, si el poder de procesamiento y almacenaje en una máquina local es muy limitado, es necesario el uso de agentes móviles, de esta manera se puede migrar a una computadora mas poderosa y lograr ejecutar la aplicación deseada

- Coordinación más sencilla. Puede ser mas simple coordinar un numero de solicitudes remotas e independientes y después solamente verificar los resultados de manera local.
- Permite cómputo asíncrono, el usuario puede activar sus agentes móviles y hacer alguna otra actividad mientras tanto y los resultados le llegaran por correo electrónico o algún otro medio, en algún tiempo posterior. Incluso puede operar aun cuando el usuario no esté "conectado".
- Pueden ir y venir dinámicamente y proporcionar servicios mucho más flexibles pueden coexistir en unidades inferiores, proporcionando más opciones para los consumidores.
- Proporciona una arquitectura flexible de cómputo distribuido única, la cual funciona de manera diferente de las arquitecturas estáticas. Esto proporciona una manera innovadora de hacer cómputo distribuido.
- Presenta una oportunidad para hacer una reestructuración radical y atractiva del proceso de diseño en general; siguiendo esto último, se dice que los agentes móviles transforman el proceso de diseño convencional.
- Aprovechamiento de la asincronía. Asincronía significa que dos actores de la comunicación no necesitan estar físicamente presentes al mismo tiempo (por ejemplo los usuarios del correo electrónico). Las ventajas de la asincronía son el mejoramiento del uso de las líneas de comunicación, la capacidad de realizar operaciones de recuperación de información más seguras y el hecho de que si el receptor está ocupado cuando la comunicación se está llevando a cabo, ésta se procesará después. Esta última propiedad es muy interesante para el cómputo móvil (PDA, laptops) donde el usuario no está permanentemente conectado. La estrategia estándar sería entonces: enviar el agente, desconectar y reconectar después.

Con respecto al uso de las líneas de comunicación, se sabe que las sesiones basadas en comunicaciones imponen una conexión permanentemente abierta entre el emisor y el receptor, esto requiere una conexión ocupada aunque nada esté pasando actualmente. Para comunicaciones de bases de datos, esto puede empeorar si las transacciones imponen algunos bloqueos, este bloqueo se mantendrá hasta que la transacción sea abortada o reanudada. Pero si un agente es despachado en una manera asíncrona, en un lugar remoto, el agente puede ejecutar un proceso asíncrono y entonces esperar por una llamada de regreso de la computadora de origen o decidir regresar por el mismo. Cuando el usuario se reconecta, recibe al agente de regreso.

Como se mencionó con anterioridad la asincronía permite realizar operaciones de recuperación de datos más seguras. Cuando una transacción es comprometida es un proceso de todo o nada, quien no ha experimentado la frustración de ver su proceso de ftp interrumpido segundos antes de terminar y tener que comenzar todo de nuevo. En el caso de agentes, una vez que el agente ha sido transferido y exitosamente recibido ya no hay de que preocuparse, ya que el agente puede pedirle al servidor remoto ser activado o reactivado las veces necesarias hasta que el trabajo haya sido terminado.

- Aprovechamiento de la autonomía. Un agente debe mostrar algo de autonomía, debe comportarse como "criatura viva" una vez que ha sido convocada. La autonomía realmente significa que no existe la necesidad de una conexión permanente entre el agente y su nodo origen, ya que en el caso de agentes móviles el agente acarrea junto con él su propio código. El agente es todavía más autónomo cuando tiene algún conocimiento de las preferencias del usuario. Esta propiedad de autonomía es muy importante ya que permite al agente trabajar por sí mismo y no requiere de una conexión permanentemente abierta.
- Aprovechamiento de las facilidades remotas. La gran contribución de los agentes móviles es ser capaces de ejecutarse en máquinas remotas. Así que pueden aprovechar las capacidades remotas en:
  - CPU, el agente es ejecutado en la máquina remota donde es más potente debido a la capacidad del CPU remoto. Esto es útil para dispositivos móviles (por ejemplo computadoras portátiles) con un CPU pobre o no disponible. Para máquinas cliente con CPU pobre, su debilidad puede ser resuelta a través de agentes.
  - Memoria, algunas operaciones pueden requerir una gran cantidad de memoria, por lo que puede ser útil tener acceso a memoria remota.
  - Multiprocesamiento, como una extensión del CPU, si el nodo remoto tiene capacidades múltiples de procesamiento, estas pueden ser usadas por el agente.
  - Multihilos, los hilos pueden ser vistos como versiones ligeras de paralelización.

  - Otros recursos, que no pueden ser hallados localmente pueden ser usados por los agentes en el nodo remoto. Por ejemplo generadores de números aleatorios coprocesadores matemáticos, hardware dedicado.

En teoría los agentes pueden ser capaces de aprovechar una gran variedad de recursos, sin embargo los aspectos de seguridad representa aún una gran barrera.

3.9.2 Desventajas

Algunas desventajas con respecto a la implementación de los agentes, son las que se presentan en algunos de los lenguajes de programación, como por ejemplo:

- La migración, que no puede ocurrir en puntos arbitrarios o requiere la captura explícita del estado de ejecución a nivel del agente.
- La comunicación entre agentes, no existe o es difícil.
- Los agentes deben ser escritos en un lenguaje específico y complejo.
- Las implementaciones solamente existen para hardware no estándar.
- El código fuente, no esta disponible para la comunidad.

Ejemplos de los lenguajes que presentan algunas de estas desventajas son:

Telescript [TEL95], desarrollado en un lenguaje orientado a objetos muy complejo, requiere hardware poderoso de propósito especial, no esta abierto a los investigadores y limita al programador a un solo lenguaje.

Tcl [TCL99], requiere que el programador explícitamente capture el estado de ejecución antes de la migración.

3.10 Seguridad

La mayoría de los autores coinciden en el que el punto más débil de los agentes móviles es la seguridad, por lo que hay que procurar los siguientes aspectos (ver Figura 21).

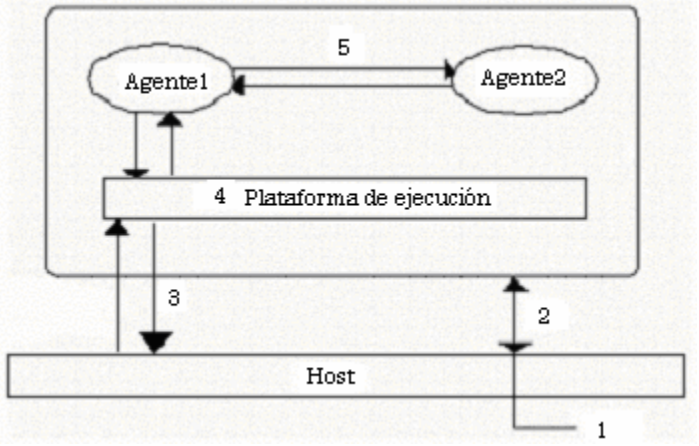

FIGURA 3.5 PUNTOS A PROCURAR SEGURIDAD

1.- Línea de transporte

2.-Proceso de autentificación y acreditación, los agentes deben ser autentificados y autorizados con algunos derechos sobre la base de su identidad

3.-Host, todas las interacciones entre la plataforma de ejecución de agentes (PEA) y el host deben ser controladas y verificadas.

4.-La plataforma de ejecución de agentes, debe ser protegida de agentes maliciosos.

5.-Comunicación entre agentes, los agentes deben estar protegidos unos de otros.

La tabla 1 muestra los aspectos a considerar, el riesgo que corren y la posible solución.

Seguridad	Peligro	Defensa
Red	Acceso a información privada encapsulada en un agente	Técnicas de criptografía, canales seguros, autentificación
Nodo	Accesos no autorizados a recursos locales	Acceso al host condicionado por el AEP
Plataforma	Interferencia con el estado de AEP por la manera de leer, escribir o ejecutar código	Agente seguro-comunicación AEP y manejo de cuentas para el acceso a los recursos de la máquina
Agente	Interferencia con el estado de un agente por leer, escribir o ejecutar código	Seguridad de la comunicación entre agentes y manejo de cuentas para el acceso a los recursos de la máquina.

TABLA 3.1 RELACIÓN DE INSEGURIDAD Y POSIBLE SOLUCIÓN.

3.11 Conclusiones

Como se ha visto los agentes son la solución para aplicaciones distribuidas de alto rendimiento, es por eso que hace necesario para ProActive contar con una infraestructura genérica que proporcione los medios para una eficiente administración de agentes.

Considerando las infraestructuras para agentes móviles propuestas, la que mejor se adapta a las necesidades de ProActive, es la presentada por Lingnau, ya que sugiere un ambiente de ejecución que en este caso caso va estar dado por Java y ProActive. El servidor de agentes propuesto se convierte en el administrador de agentes para la infraestructura genérica de agentes bajo ProActive.

En el siguiente capítulo, se presentará el desarrollo y modelado de la infraestructura genérica para agentes móviles en ProActive.

Capítulo 4 Diseño de la infraestructura genérica para agentes móviles

4.1 Introducción

El presente capítulo está dedicado a presentar la solución y al modelado de la infraestructura genérica para agentes móviles a través del lenguaje unificado de modelado (UML). Se usó UML debido a que permite generar diseños que capturan las ideas de manera convencional y fácil de comprender para comunicarlas de manera eficiente a otras personas.

4.2 Diseño

Una infraestructura genérica consiste de primitivas básicas que permiten el funcionamiento de una actividad, en el caso particular de los agentes, se proporcionan las primitivas mínimas necesarias para la administración de lo agentes, generados bajo el ambiente de ProActive.

La infraestructura genérica para agentes propuesta en este trabajo de tesis, sienta las bases para la administración eficiente y transparente de agentes móviles permitiendo de esta manera el desarrollo de aplicaciones distribuidas con agentes móviles. Considerando como operaciones básicas, el lanzamiento de agentes, recuperación de información del estado y ubicación actual del agente, obtención del resultado, si el cometido del agente fue realizado exitosamente o no, suspender y un momento dado reanudar la actividad del agente, cambiar el itinerario del agente definido por defecto.

La infraestructura está basada en la propuesta por Lingnau y colegas en [LDD95], en la que considerando las especificaciones de ejecución requeridas por ProActive (misma versión de Java y ProActive, en las máquinas involucradas), es posible adaptarla a los requerimientos de la infraestructura genérica para agentes.

La Interfaz Gráfica del Agente (Graphical Interface Agent-GIA), es la interfase gráfica para el servidor de agentes, a través del cual es posible la administración eficiente de los agentes. Por ser un objeto activo, el GIA tiene la capacidad de migrar, esto implica que no es necesario que se encuentre ejecutando en cada computadora involucrada en el cómputo.

El ambiente de ejecución está dado por Java y ProActive, en donde el servidor esta en un host, generando agentes a petición del usuario. Tanto el GIAcomo el agente mismo son objetos activos con la capacidad de migrar de un host a otro. El agente mantiene una comunicación bilateral con el servidor agentes y/o con otros agentes ya sean local o remotamente (ver Figura 4.1).

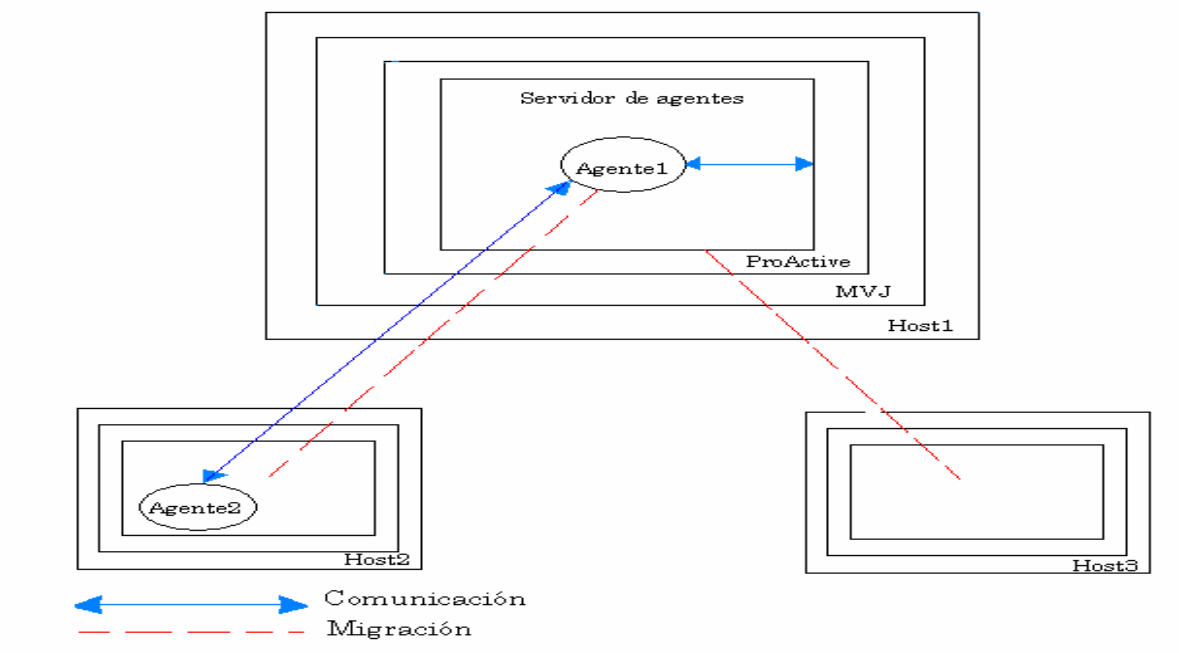

FIGURA 4.1 ARQUITECTURA DE LA INFRAESTRUCTURA GENÉRICA PARA AGENTES

Las primitivas aportadas por la infraestructura forman parte y están diseñadas a partir de la biblioteca ProActive.

Una de las piezas principales de la infraestructura genérica es sin duda el agente, para ProActive el agente es en esencia un objeto activo (ver Figura 4.2) al que por sus características tanto de ubicación y actividad transparente se le ha adicionado nuevos métodos, creando una nueva metaclase *Agent* con el fin de estandarizar la programación de los agentes y permitir la comunicación con el administrador de agentes tal como se muestra en la Figura 4.3.

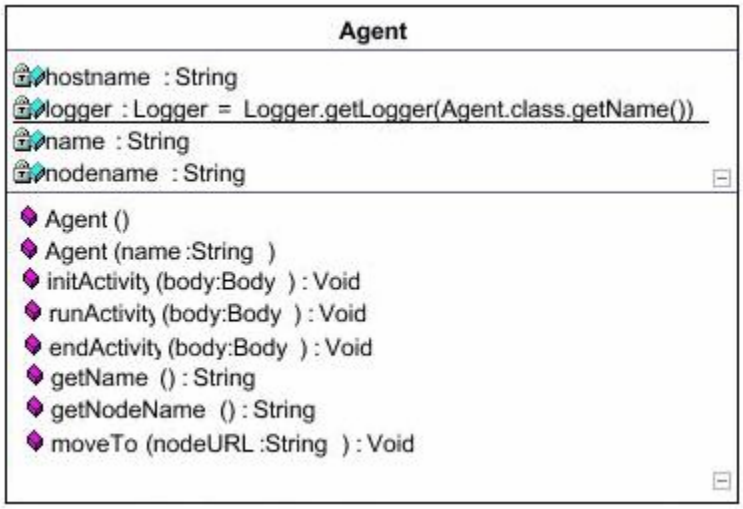

FIGURA 4.2 AGENTE PROACTIVE

FIGURA 4.3 AGENTE ESTANDARIZADO

4.3 Modelado de la infraestructura genérica para agentes móviles en ProActive

4.3.1UML

Uno de los problemas más comunes en el desarrollo de software es que la mayoría de programadores y analistas es reacia a documentar las decisiones tomadas, bien por falta de tiempo o por desconocimiento de las bondades de esta actividad, y cuando se hace, suele ser algo incompleto, no actualizado y poco consistente, ya que cada miembro del equipo utiliza una serie de símbolos familiares para él pero no para los demás.

En este contexto, UML surge como respuesta al primer problema reseñado para contar con un lenguaje estándar para escribir planos de software

El Lenguaje Unificado de Modelado, UML [BOO99] es una notación estándar para el modelado de sistemas software. UML es el resultado de una propuesta de estandarización promovida por el consorcio OMG (Object Management Group), del cual forman parte las empresas más importantes que se dedican al desarrollo de software, en 1996.

UML representa la unificación de las notaciones de los métodos Booch, Objectory (Ivar Jacobson) y OMT (James Rumbaugh) siendo su sucesor directo y compatible. Igualmente, UML incorpora ideas de otros metodólogos entre los que podemos incluir a Peter Coad, Derek Coleman, Ward Cunningham, David Harel, Richard Helm, Ralph Johnson, Stephen Mellor, Bertrand Meyer, Jim Odell, Kenny Rubin, Sally Shlaer, John Vlissides, Paul Ward, Rebecca Wirfs-Brock y Ed Yourdon.

Un enfoque sistemático permite construir modelos de una forma consistente demostrando la utilidad del modelado en sistemas de todo tamaño, independientemente de que se trate de un programa de cincuenta, cien líneas ó de cientos de desarrolladores trabajando y compartiendo información, el hecho de compartir información sobre las decisiones tomadas, es vital no sólo durante el desarrollo del proyecto, sino una vez finalizado éste, cuando se requiere algún cambio en el sistema.

4.3.2 Diagramas de casos de uso

4.3.2.1 Diagrama de caso de uso GIA

En la Figura 4.4 se modelan las acciones del sistema GIA, en donde el actor es el usuario final haciendo uso del sistema GIA, solicitando los servicios de lanzamiento de agentes, información general de los agentes, resultados de los agentes de las actividades de los agentes, la suspensión de la actividad del agente, reanudando la actividad del agente.

A su vez el sistema GIA canaliza las solicitudes del usuario hacia el agente.

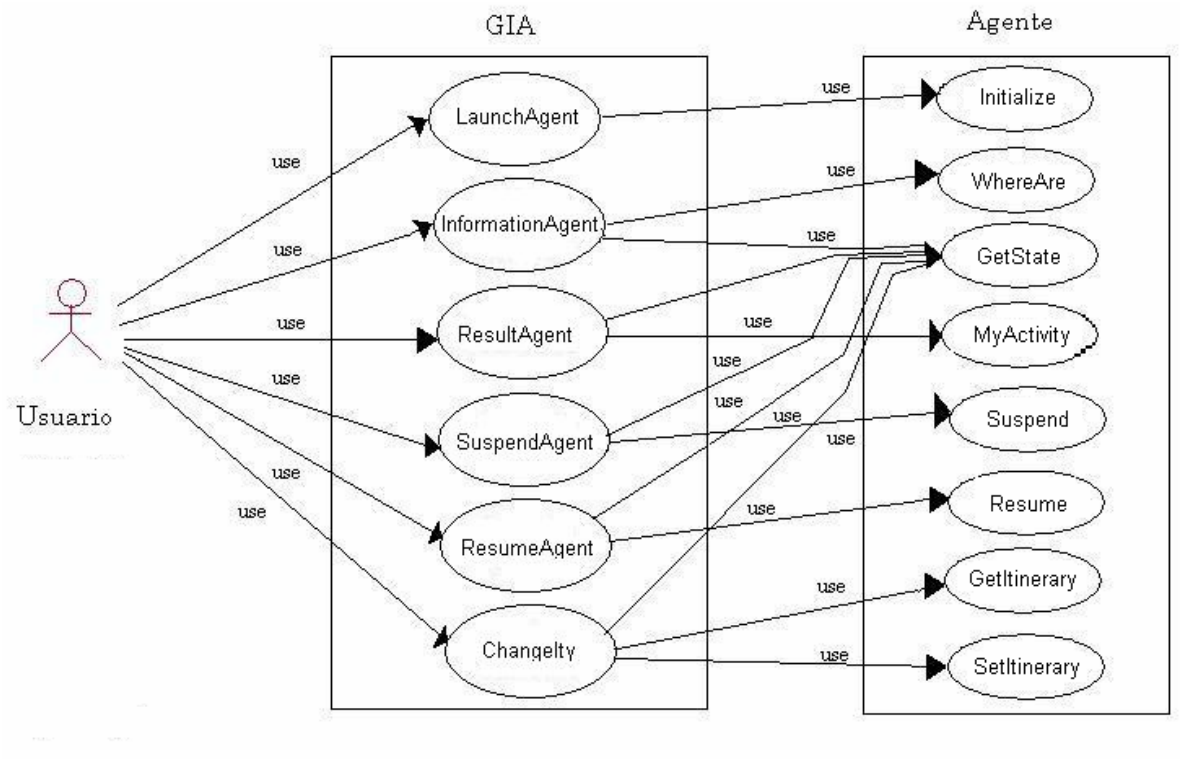

FIGURA 4.4 CASO DE USO DE GIA

Los diagramas siguientes muestran las relaciones para cada uno de los casos de uso de GIA.

4.3.2.2 Diagrama de caso de uso LaunchAgent

El caso de uso LaunchAgent (lanzar agente) tal como se muestra en la Figura 4.5, lleva consigo tres parámetros (name, parameter, itinerary), para lograr su cometido, es necesario un proceso de verificación (Verification) que valide si la clase del agente a lanzar existe en el ambiente de ejecución (ProActive).

Si la clase existe la clase se carga en memoria (Load) y se agrega el agente a una lista circular (addAgent), proporcionándole un identificador único. Si no fuera posible agregar el agente, la clase del agente es descargado (Unload).

En el caso de que la clase no existiera o de que no se pudiera cargar, el usuario será informado.

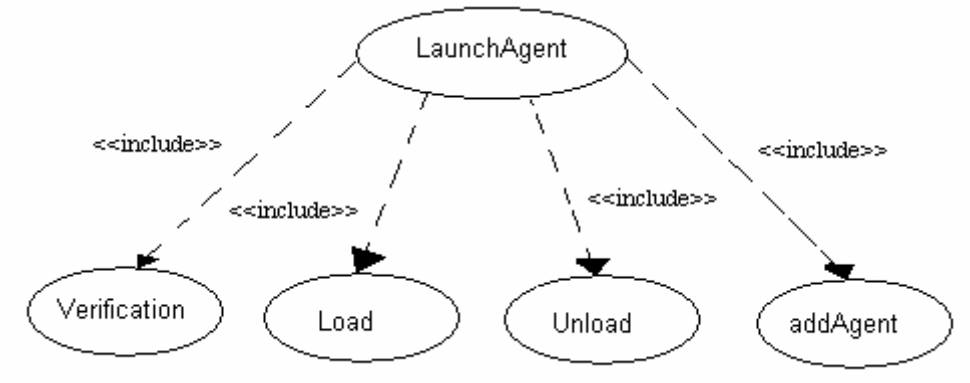

FIGURA 4.5 CASO DE USO LAUNCHAGENT

4.3.2.3 Diagrama de caso de uso InformationAgent

El caso de uso InformationAgent (Información del agente) tiene como parámetro la lista de agentes, y el proceso consiste de verificar que existan agentes en la lista circular, si existen, a través de un ciclo realiza una conexión (Connection) con cada una ellos para solicitar su ubicación actual (WhereAre) y su estado (GetState) y mostrarla en una tabla con sus respectivos identificadores (ver Figura 4.6).

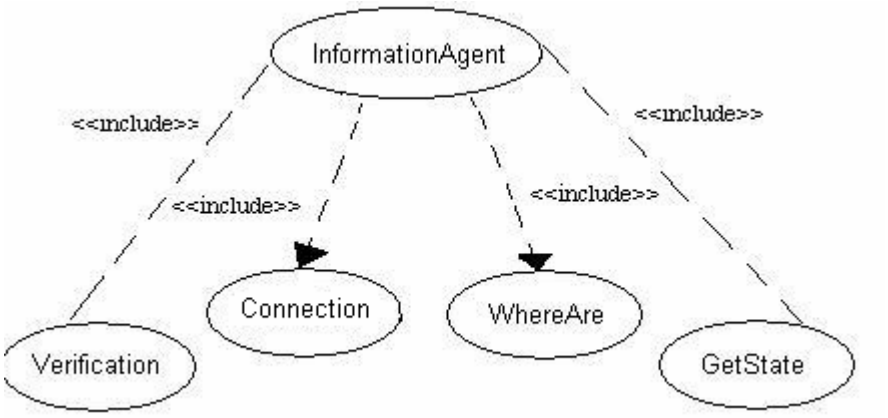

FIGURA 4.6 CASO DE USO INFORMATIONAGENT

4.3.2.4 Diagrama de caso de uso ResultAgent

El caso de uso ResultAgent tiene como parámetro el identificador del agente (id), para obtener su estado que es evaluado para conocer si el agente ya concluyó su actividad ya sea exitosamente o no, si este es el caso se solicita la información obtenida al agente (MyActivity), para mostrarla al usuario (ver Figura 4.7).

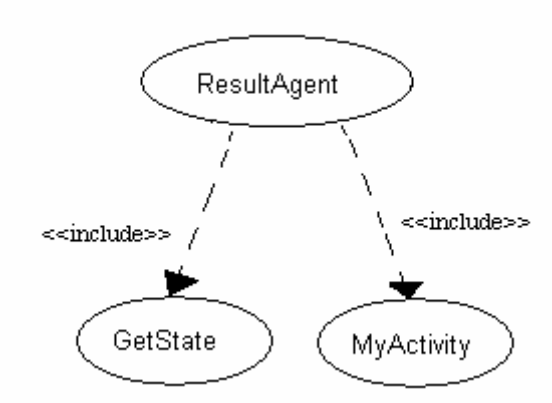

FIGURA 4.7 CASO DE USO RESULTAGENT

4.3.2.5 Diagrama de caso de uso SuspendAgent

En el caso de uso ResultAgent con su respectivo parámetro (id), es necesario obtener el estado del agente, por que si el agente ya terminó o el agente ya está suspendido no va ser posible suspender su actividad. Si este esta en actividad, se suspende la misma (Suspend), tal como se muestra en la Figura 4.8.

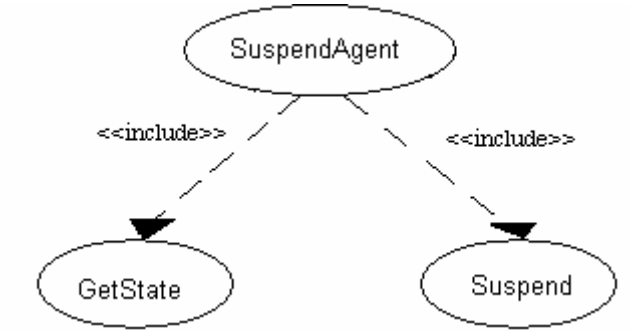

FIGURA 4.8 CASO DE USO SUSPENDAGENT

4.3.2.6 Diagrama de caso de uso ResumeAgent

En el caso de uso ResumeAgent con su respectivo parámetro (id), es necesario obtener el estado del agente, por que si el agente ya terminó ò si el agente está activo no va ser posible reanudar su actividad. Si éste esta suspendido, se reactiva su actividad (Resume), tal como se muestra en la Figura 4.9.

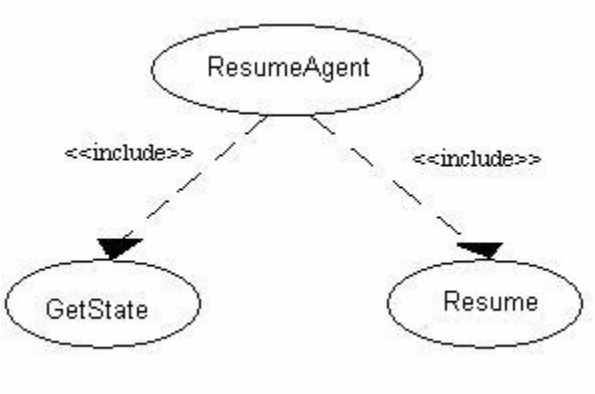

FIGURA 4.9 CASO DE USO RESUMEAGENT

4.3.2.7 Diagrama de caso de uso ChangeIty

Y por último el caso de uso ChangeIty con el parámetro id, se obtiene el estado del agente con la finalidad de conocer si esta ya ha finalizado su actividad, no es posible cambiar su itinerario. Si el agente está en actividad o suspendido se obtiene el itinerario (GetItinerary) del agente ya sea para agregar un nuevo host (AddNewHost) o remover (RemoveHost) uno, en cualquiera de los casos se confirman los cambio, si este es positivo se establece un nuevo itinerario (SetItinerary). Si la confirmación es negativa se continúa con el itinerario original (ver Figura 4.10).

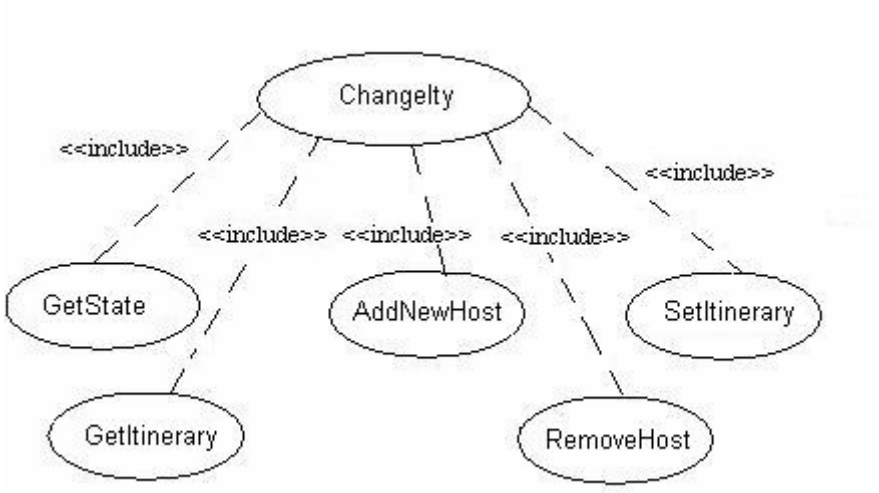

FIGURA 4.10 CASO DE USO CHANGEITY

4.3.3 Identificación de clases

El siguiente paso es la identificación de las clases a partir del análisis de los casos de uso (para cuestiones de estandarización en la programación se aclara que GIA es igual al termino Gia utilizado a partir de la identificación de clases).
Las clases que conforman la infraestructura son las siguientes:
Gia, es la clase que va tener el control de las operaciones a realizar con el agente (ver Figura 4.11).
GiaApplet, es la interfaz gráfica a través de la cual el usuario final va a poder interaccionar con el agente por medio de Gia.
Agent es la clase estándar a partir de la cual es posible programar agentes para un fin particular, capaces de interaccionar con el Gia.

Las siguientes clases son colaboradoras, en donde
MyTableModel, se obtendrá una instancia de esta clase, para la presentación de la información de los agentes (id, estado, ubicación).
AgentListModel, con la colaboración de esta clase, los agentes generados tendrán sus referencias en una lista circular, con un identificador único.
AgentWrapper, esta clase es de gran utilidad, por que a través de ella se podrá envolver las operaciones de cambio de itinerario del agente seleccionado.

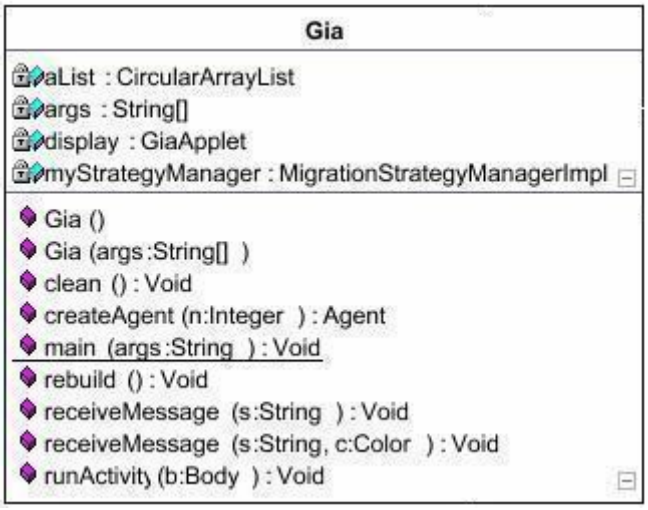

FIGURA 4.11 IDENTIFICACIÓN DE CLASE LA GIA

GiaMessageReceiver, es una clase identificada para el envío y recepción de mensajes entre Gia y el agente (ver Figura 4.12).

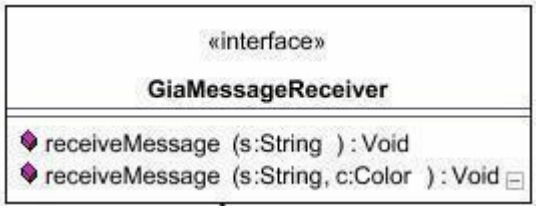

FIGURA 4.12 IDENTIFICACIÓN DE LA CLASE GIAMESSAGERECEIVER

GiaApplet, es la interfaz gráfica a través de la cual el usuario final va a poder interaccionar con el agente por medio de la clase controladora Gia (ver Figura 4.13).

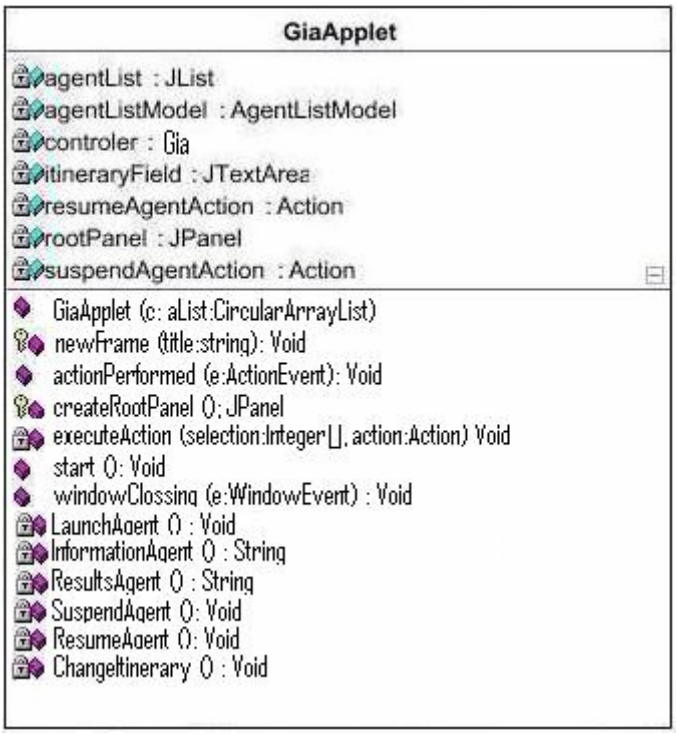

FIGURA 4.13 IDENTIFICACIÓN DE LA CLASE GIAAPPLET

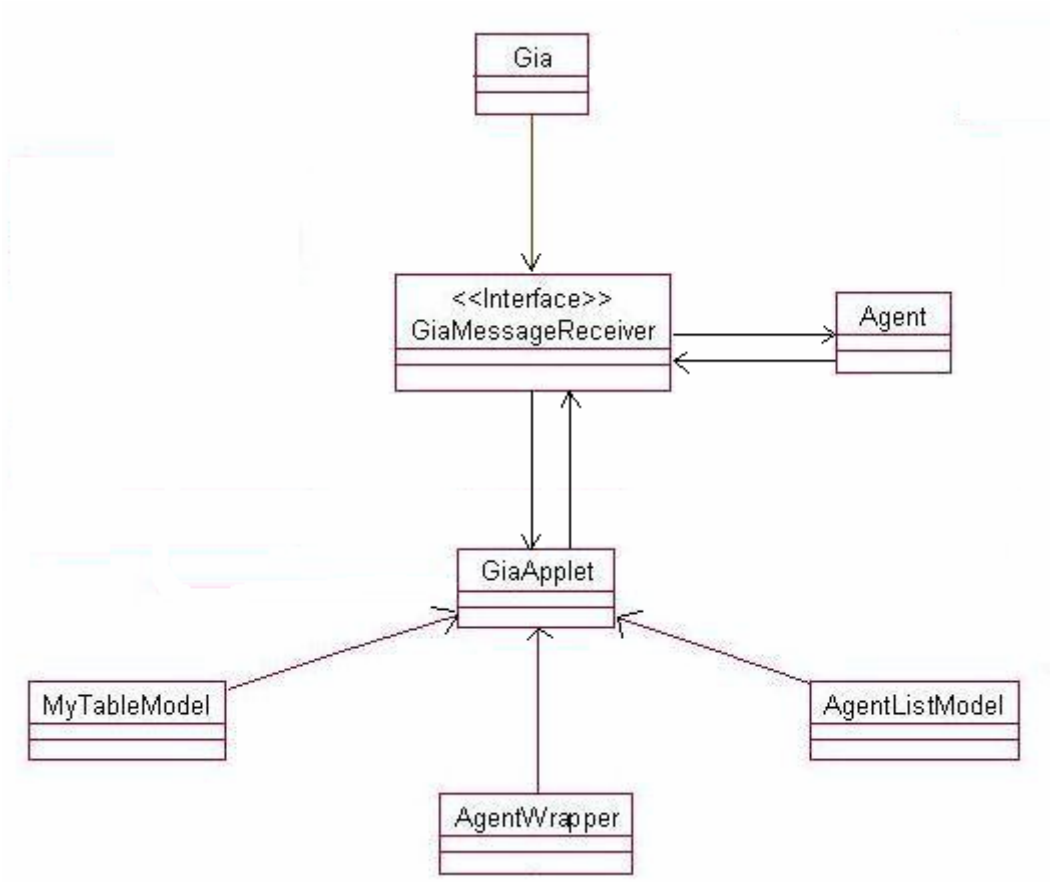

FIGURA 4.14 ASOCIACIÓN DE CLASES

4.3.4 Diagramas de clases

En este diagrama se muestra la manera en como se relacionan las clases antes identificadas, ahora con sus atributos y métodos particulares, así como la herencia de comportamientos, y las responsabilidades de cada una de ellas (ver. Figura 4.15).

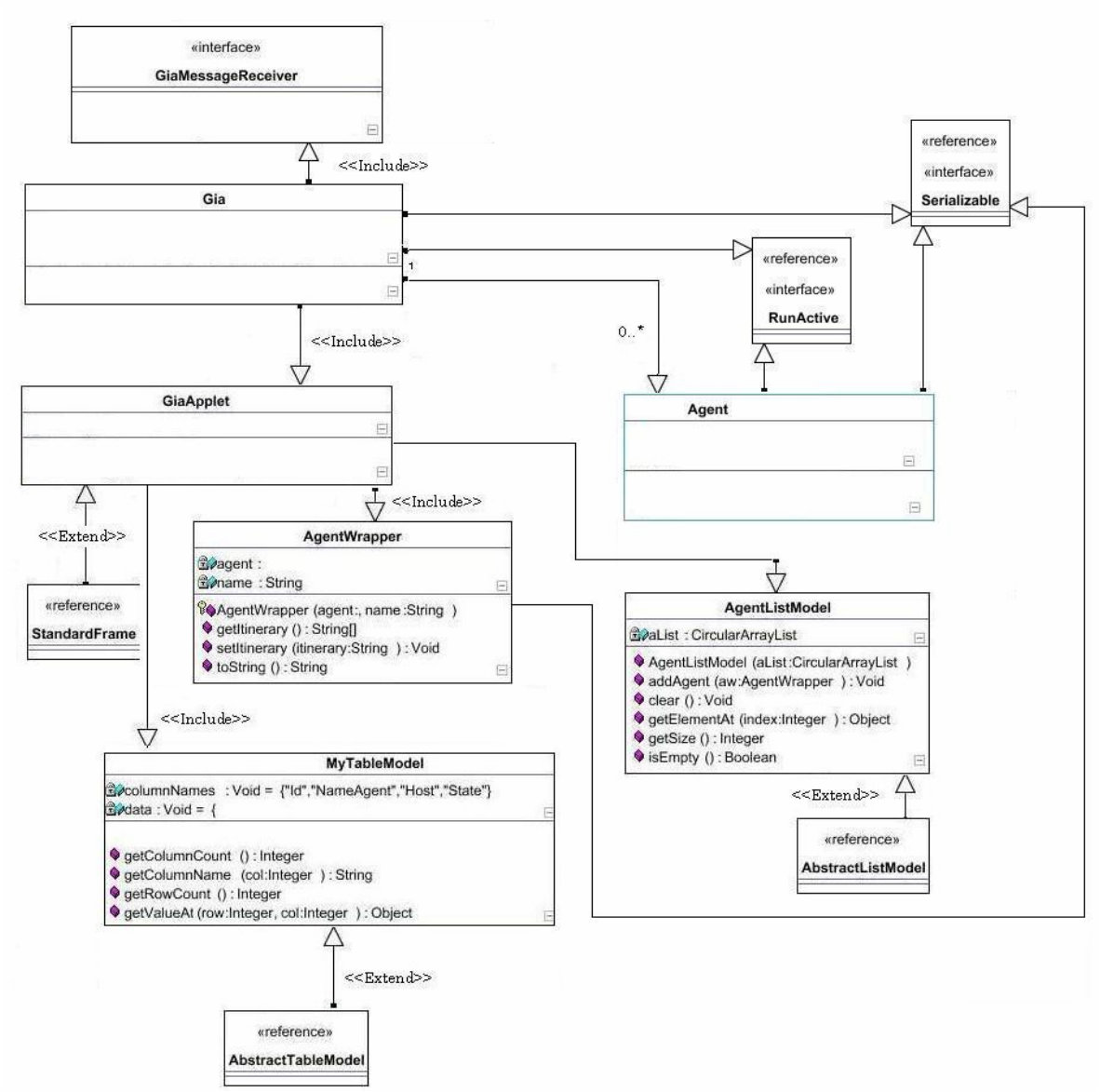

FIGURA 4.15 DIAGRAMA DE CLASES

4.3.5 Diagramas de actividades

4.3.5.1 Diagrama de actividades

Con el diseño de estos diagramas se representan las actividades que ocurren dentro de cada caso de uso a detalle (ver Figura 4.16, 4.17, 4.18, 4.19, 4.20, 4.21).

4.3.5.1 Diagrama de actividades LaunchAgent

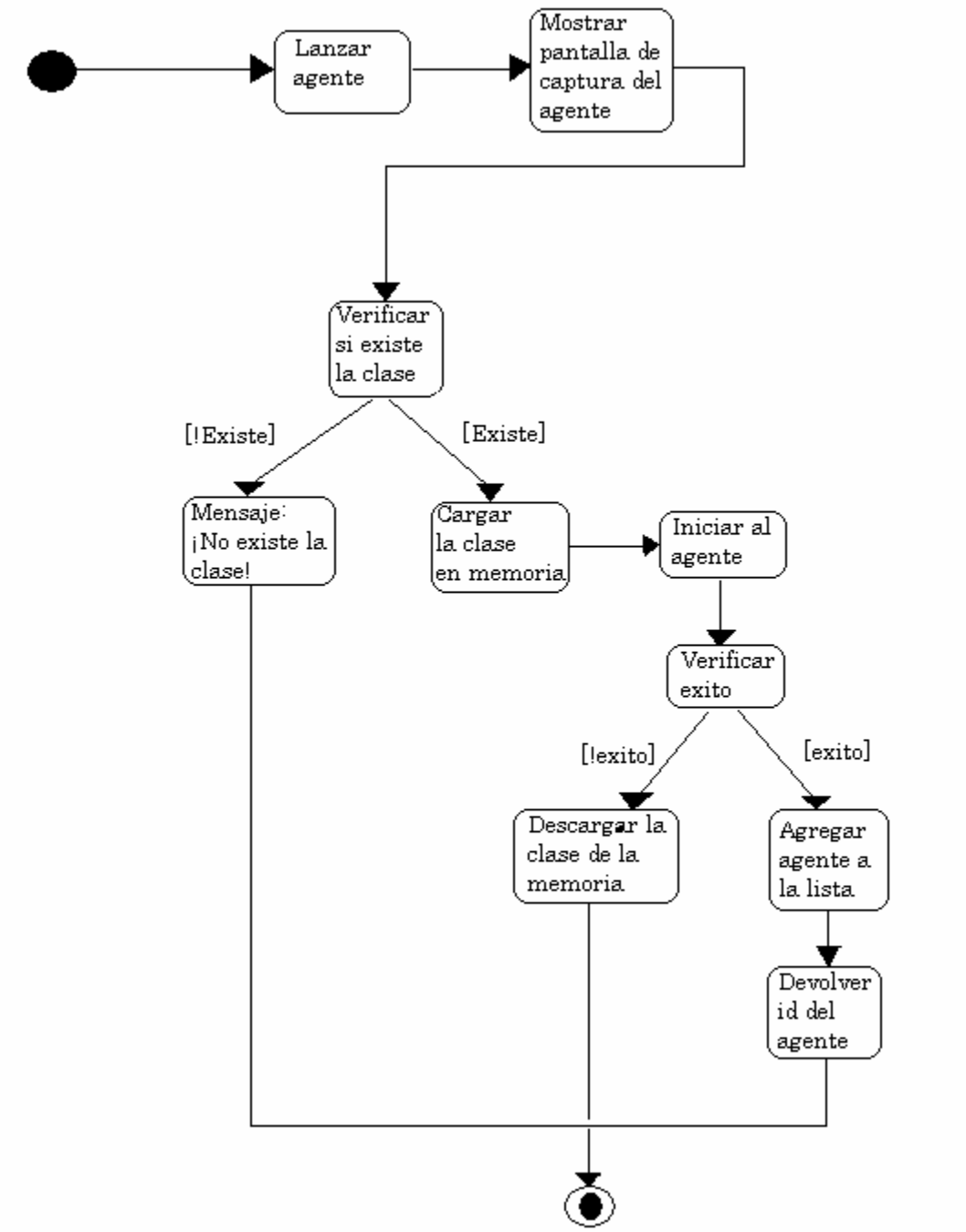

FIGURA 4.16 DIAGRAMA DE ACTIVIDADES DE LANZAR AGENTE

4.3.5.2 Diagrama de actividades InforAgent

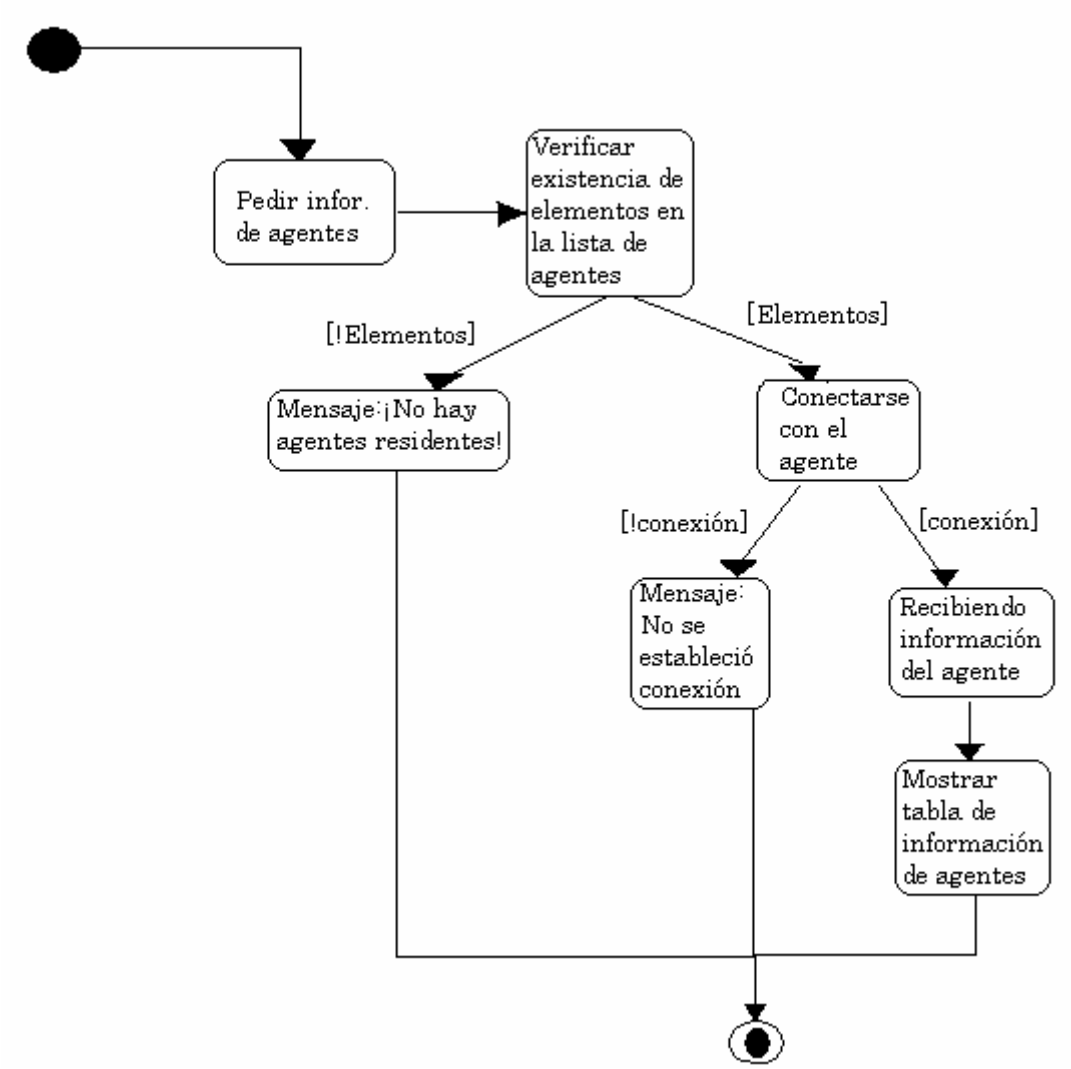

FIGURA 4.17 DIAGRAMA DE ACTIVIDADES DE SOLICITAR INFORMACIÓN

4.3.5.3 Diagrama de actividades ResultAgent

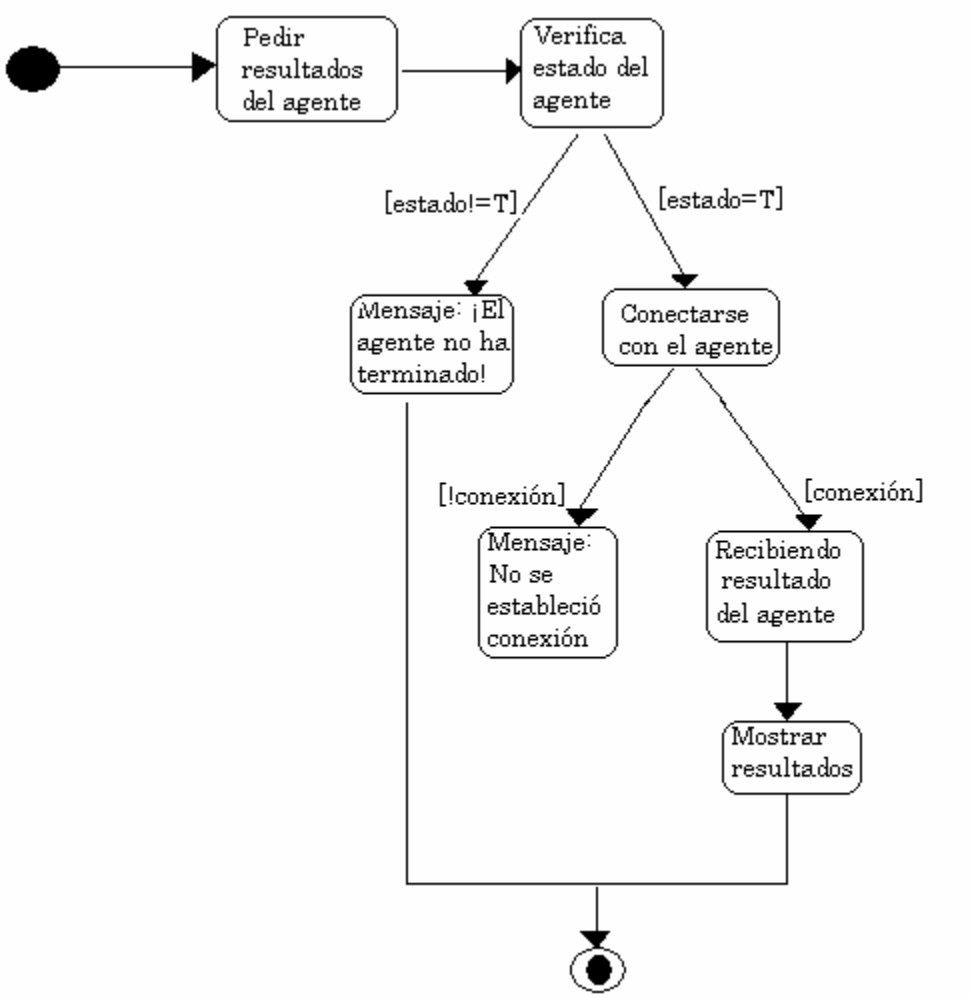

FIGURA 4.18 DIAGRAMA DE ACTIVIDADES DE SOLICITAR RESULTADOS

4.3.5.4 Diagrama de actividades SuspendAgent

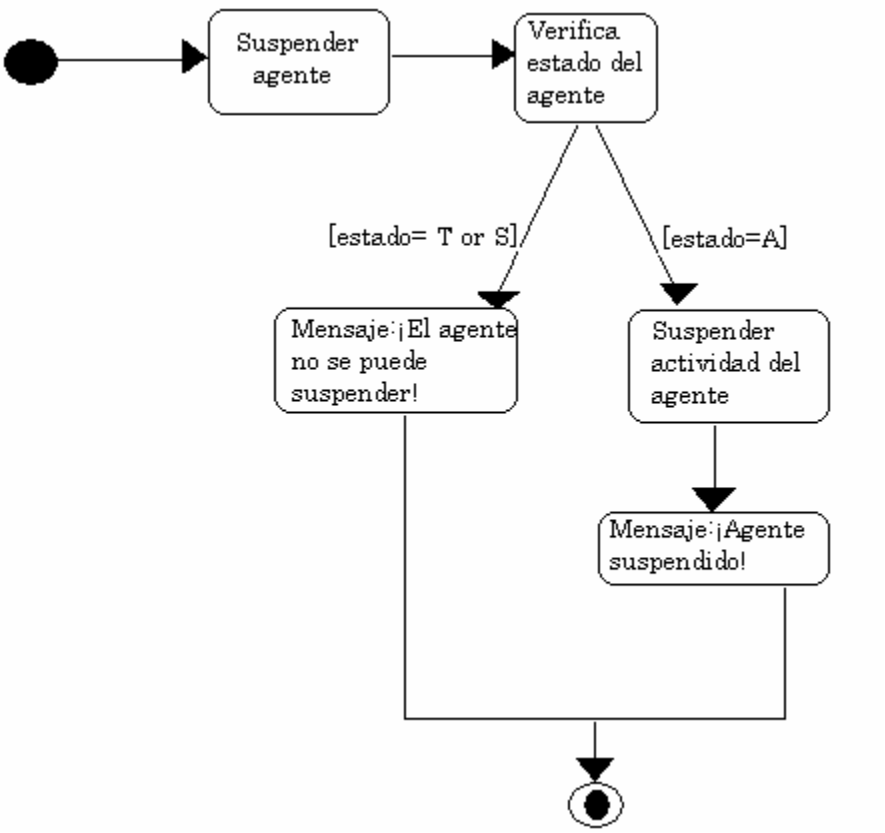

FIGURA 4.19 DIAGRAMA DE ACTIVIDADES DE SUSPENDER

4.3.5.5 Diagrama de actividades ResumeAgent

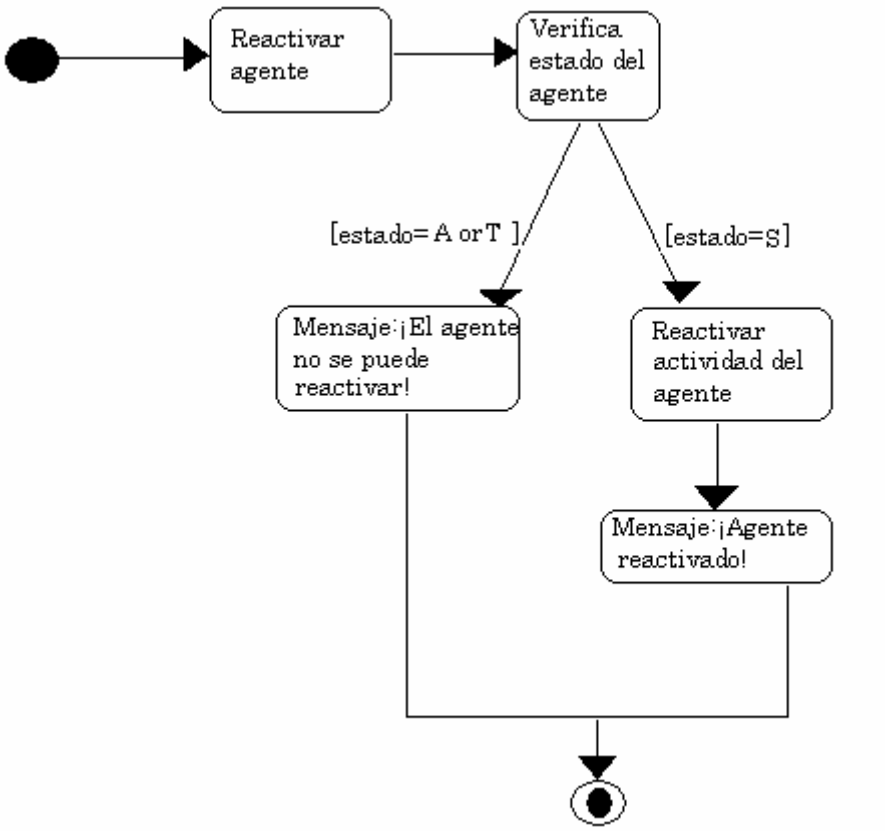

FIGURA 4.20 DIAGRAMA DE ACTIVIDADES DE REANUDAR

4.3.5.5 Diagrama de actividades Changelty

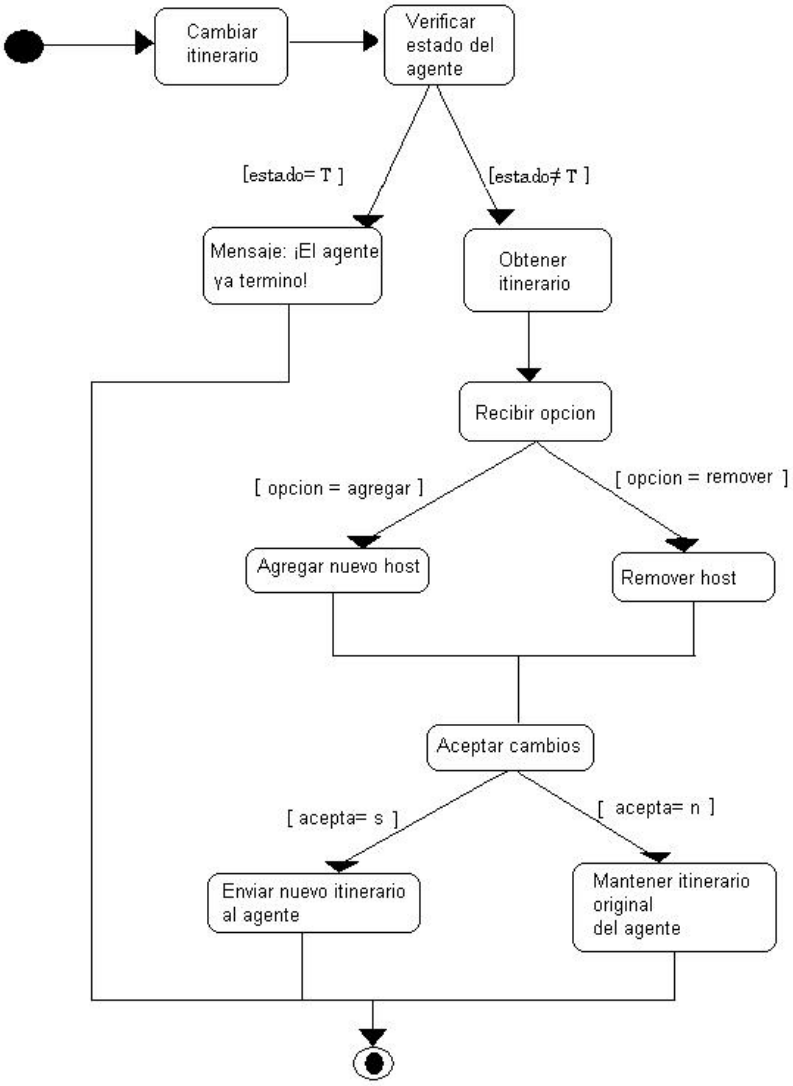

FIGURA 4.21 DIAGRAMA DE ACTIVIDADES DE CAMBIAR ITINERARIO

4.3.6 Diagrama de secuencias

Se muestra a través de estos diagramas la forma en que los objetos colaboran entre sí, destacando la sucesión de las interacciones, por cada método identificado (ver Figura 4.22, 4.23, 4.24, 4.25., 4.26, 4.27).

4.3.6.1 Diagrama de secuencias LaunchAgent

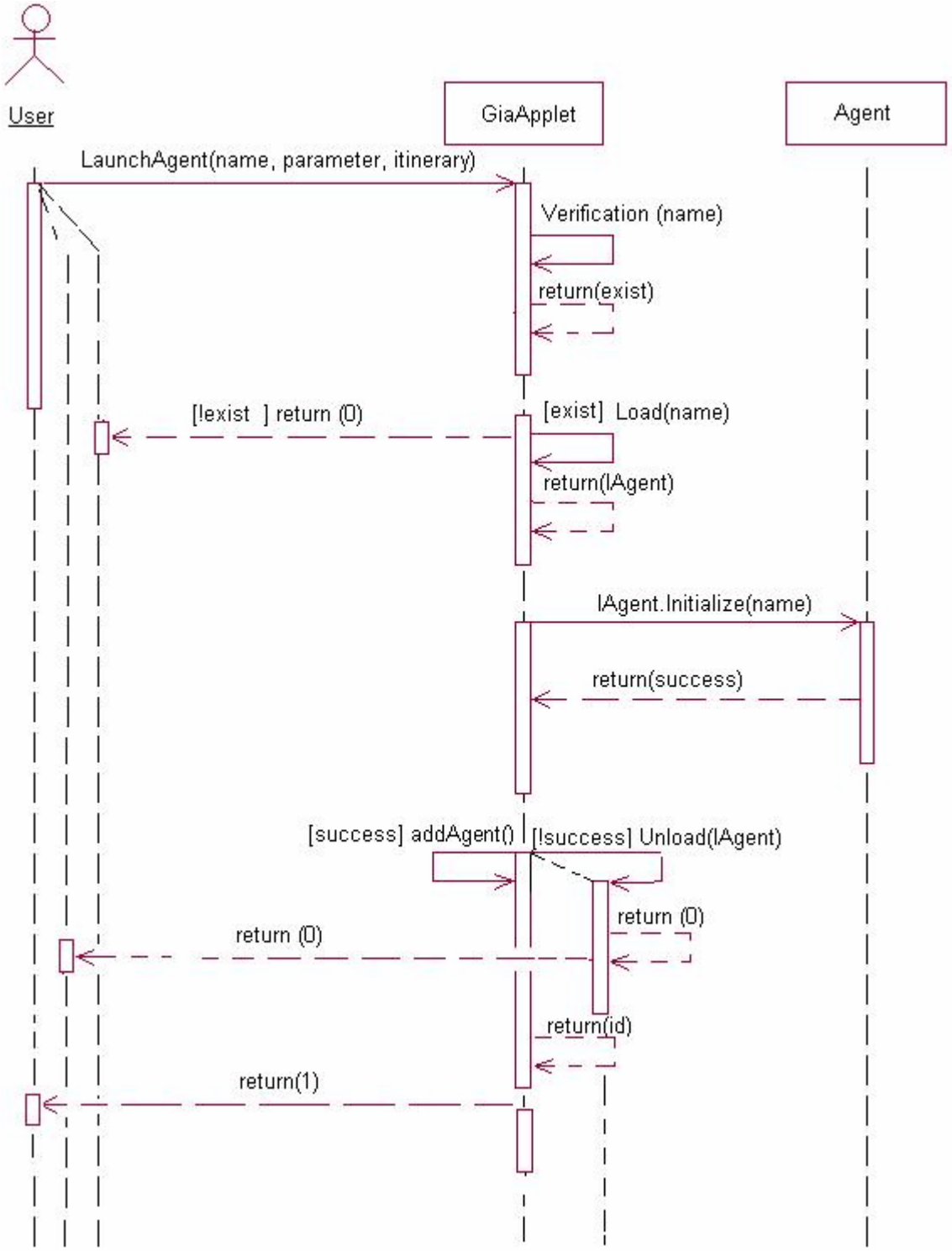

FIGURA 4.22 DIAGRAMA DE SECUENCIAS DEL MÉTODO LAUNCHAGENT

4.3.6.2 Diagrama de secuencias InforAgent

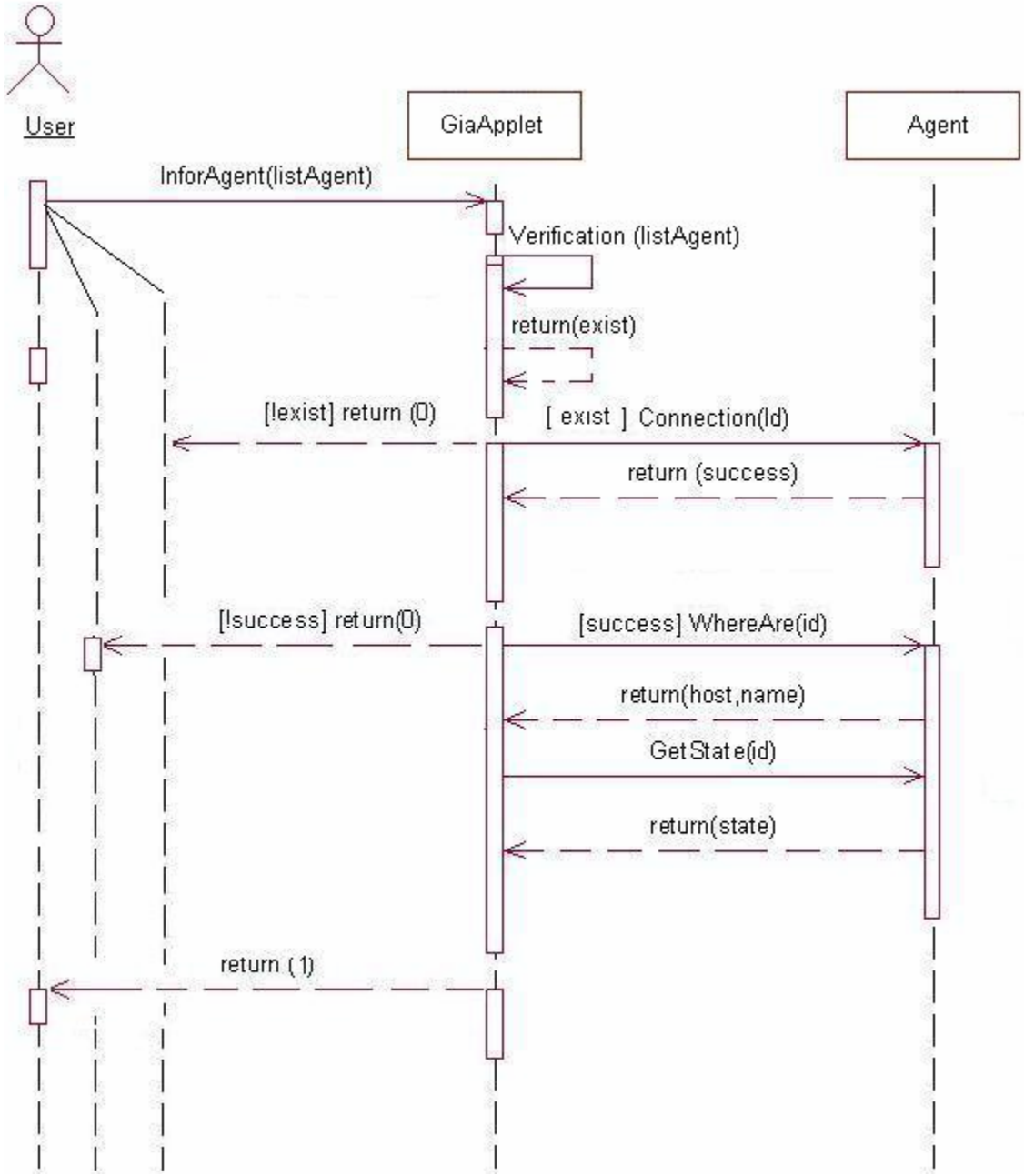

FIGURA 4.23 DIAGRAMA DE SECUENCIAS MÉTODO INFORAGENT

4.3.6.3 Diagrama de secuencias ResultAgent

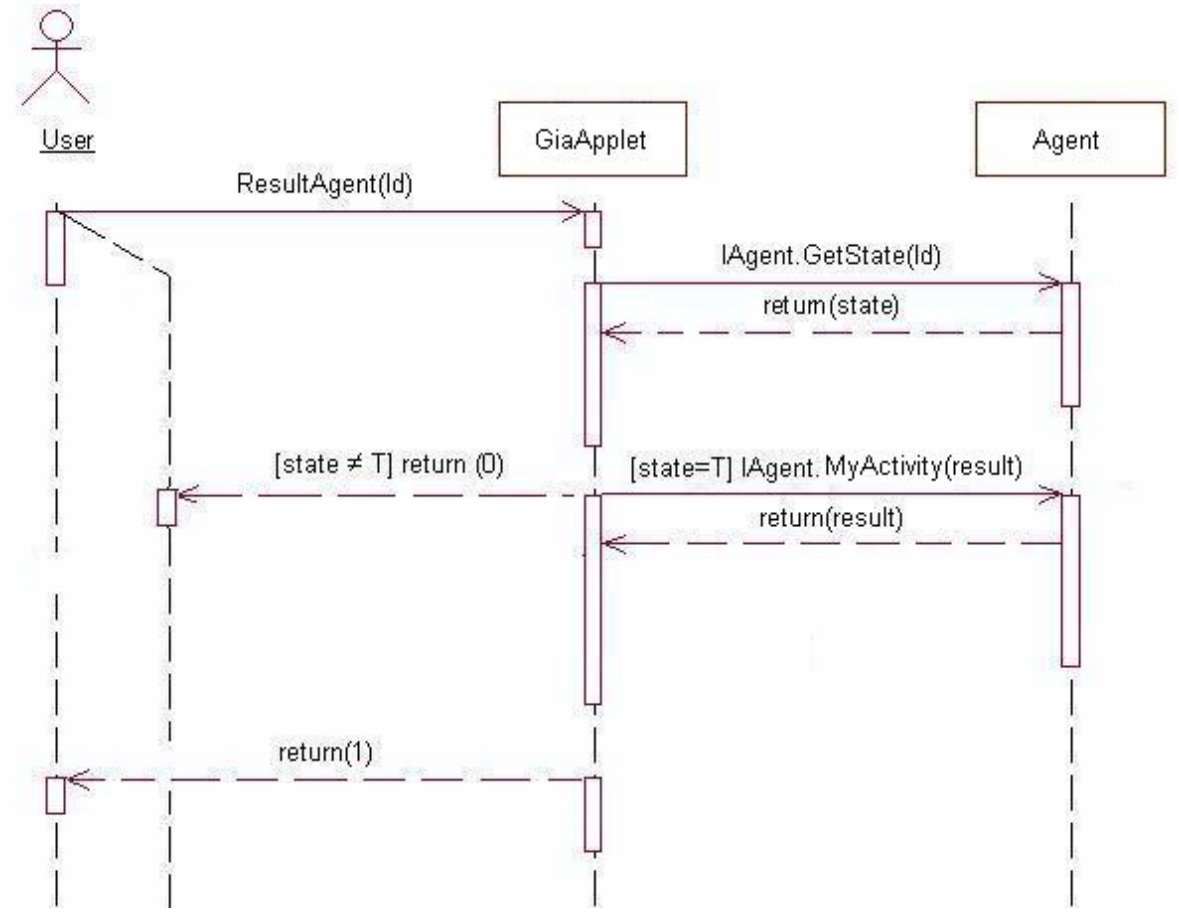

FIGURA 4.24 DIAGRAMA DE SECUENCIAS DEL MÉTODO RESULTAGENT

4.3.6.4 Diagrama de secuencias SuspendAgent

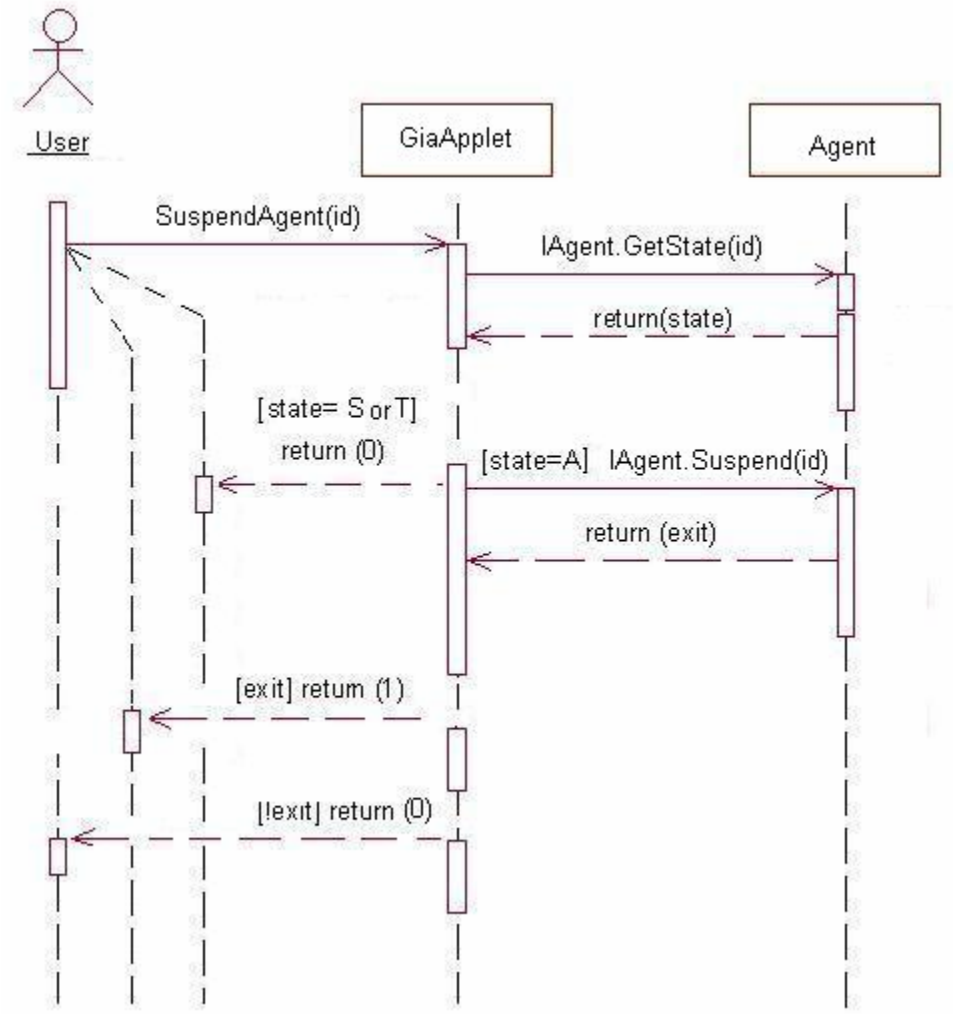

FIGURA 4.25 DIAGRAMA DE SECUENCIAS DE MÉTODO SUSPENDAGENT

4.3.6.5 Diagrama de secuencias ResumeAgent

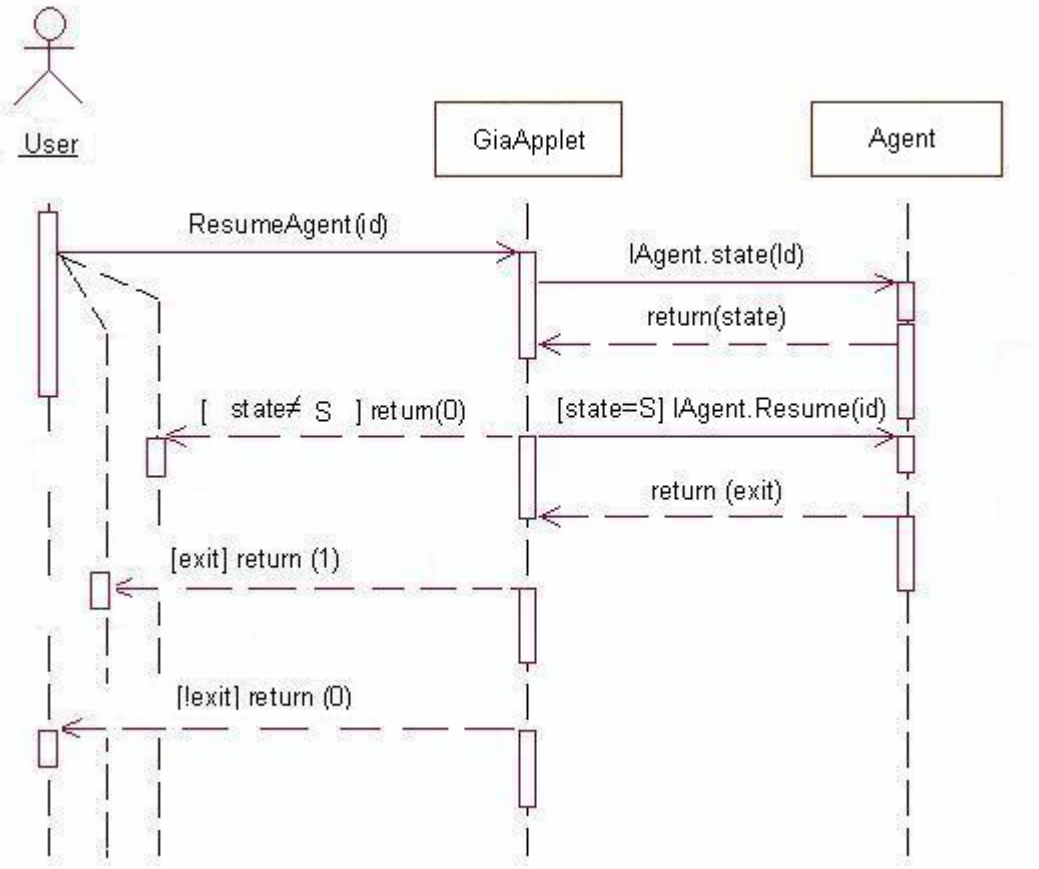

FIGURA 4.26 DIAGRAMA DE SECUENCIAS MÉTODO RESUMEAGENT

4.3.5.6 Diagrama de secuencias Changelty

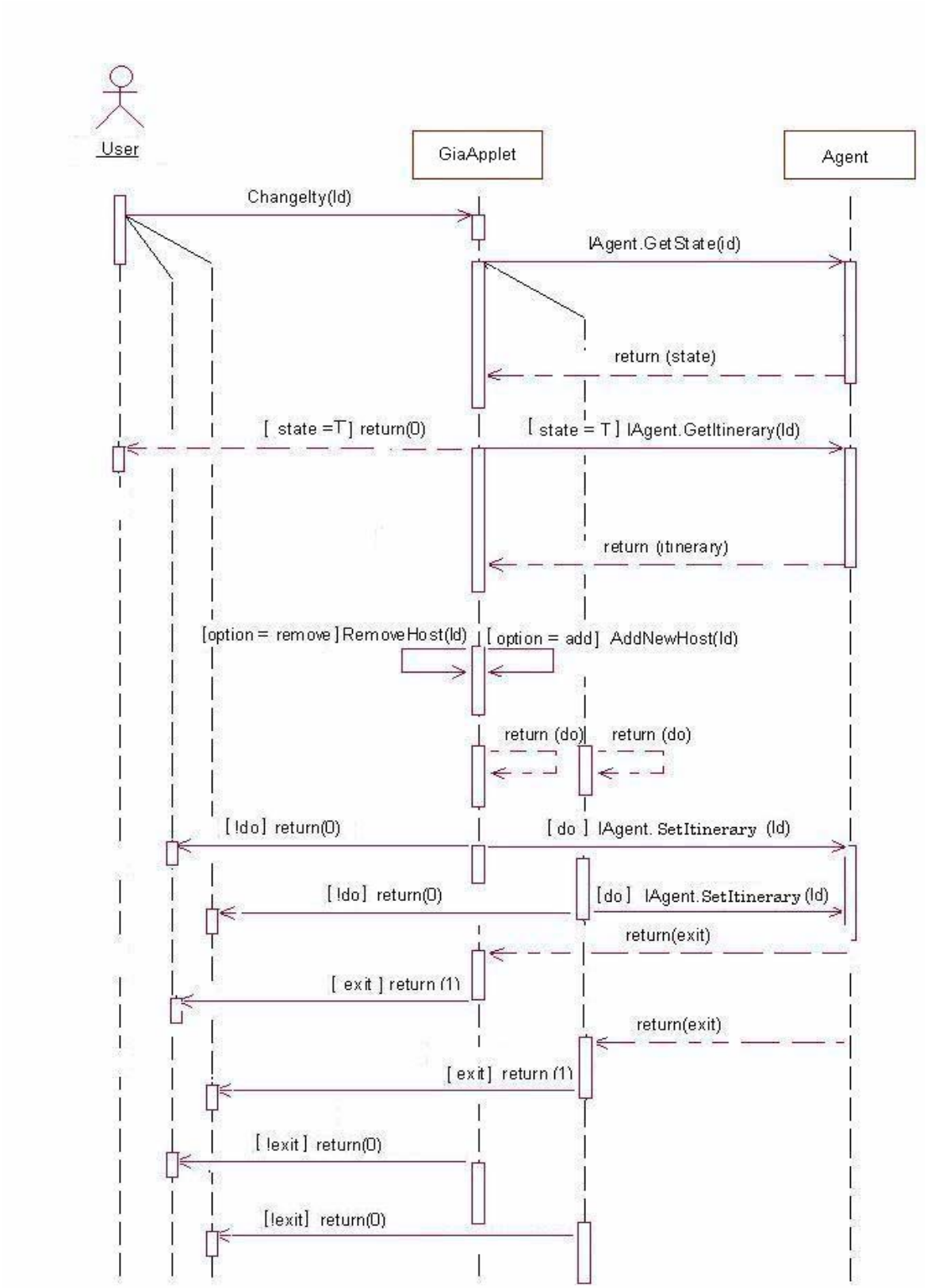

FIGURA 4.27 DIAGRAMA DE SECUENCIAS MÉTODO CHANGEITY

4.3.6 Diagrama de componentes

Este tipo de diagramas muestra la organización y las dependencias entre un conjunto de componentes dados, al conjunto de componentes se le llama paquete.

Tal como se muestra en la Figura 4.28, los componentes Gia.java, GiaMessageReceiver.java y GiaApplet.java forman parte del paquete gia. El componente GiaApplet.java interacciona con el agente y la comunicación se realiza a través del componente GiaMessageReceiver. El paquete gia se relaciona por completo con el paquete ProActive ya hace uso de los componentes que contiene.

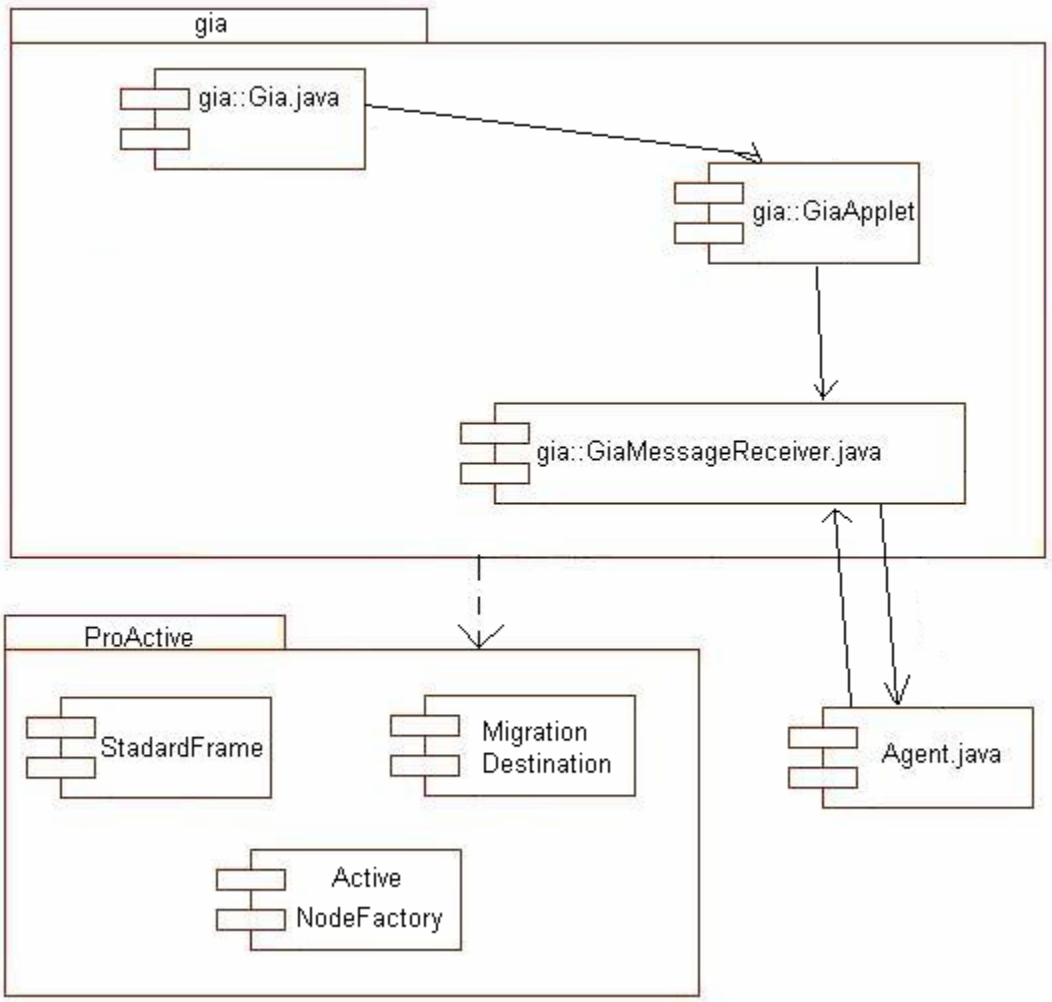

FIGURA 4.28 DIAGRAMA DE COMPONENTES

4.3.7 Diagrama de despliegue

Este tipo de diagrama muestra los dispositivos que se encuentran en el sistema y su distribución en el mismo (ver Figura 4.29).

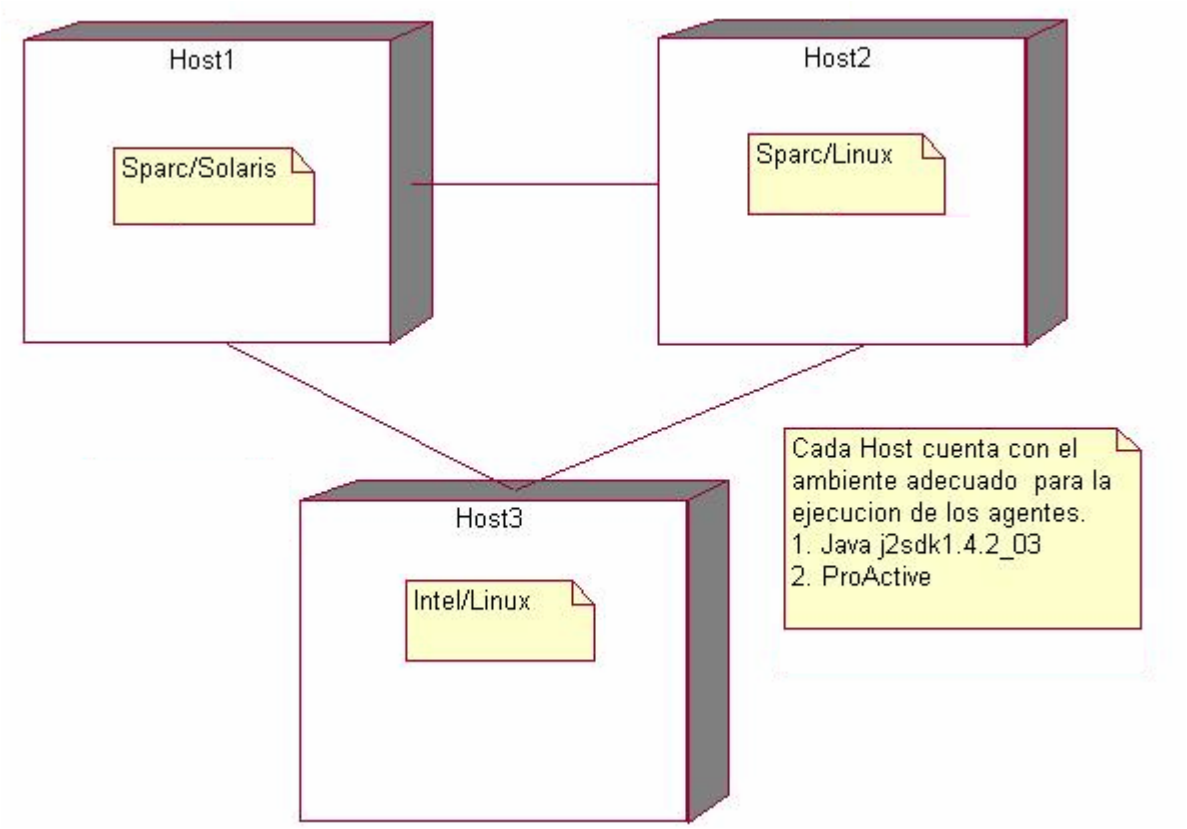

FIGURA 4.29 DIAGRAMA DE DESPLIEGUE

4.3.8 Diagrama de la interfaz gráfica

Este diagrama muestra el modelado de la ventana principal de la interfaz (ver Figura 4.30).

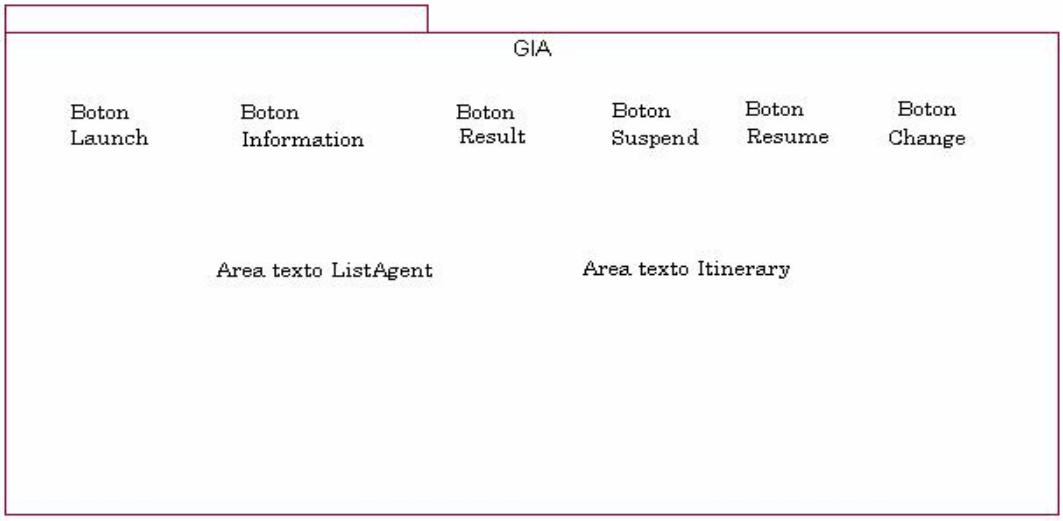

FIGURA 4.30 MODELADO DE LA VENTANA PRINCIPAL DE LA INTERFAZ

4.4 Prototipo

La ventana actividad de la interfase esta conformada tal como se observa en la Figura 4.31, en donde es posible observar los botones que conllevan a una acción, un área para la visualización de la lista de los agentes, un área de texto que muestra el itinerario de un agente en particular. Por ultimo un área de mensajes donde se visualizan los mensajes enviados del agente a Gia.

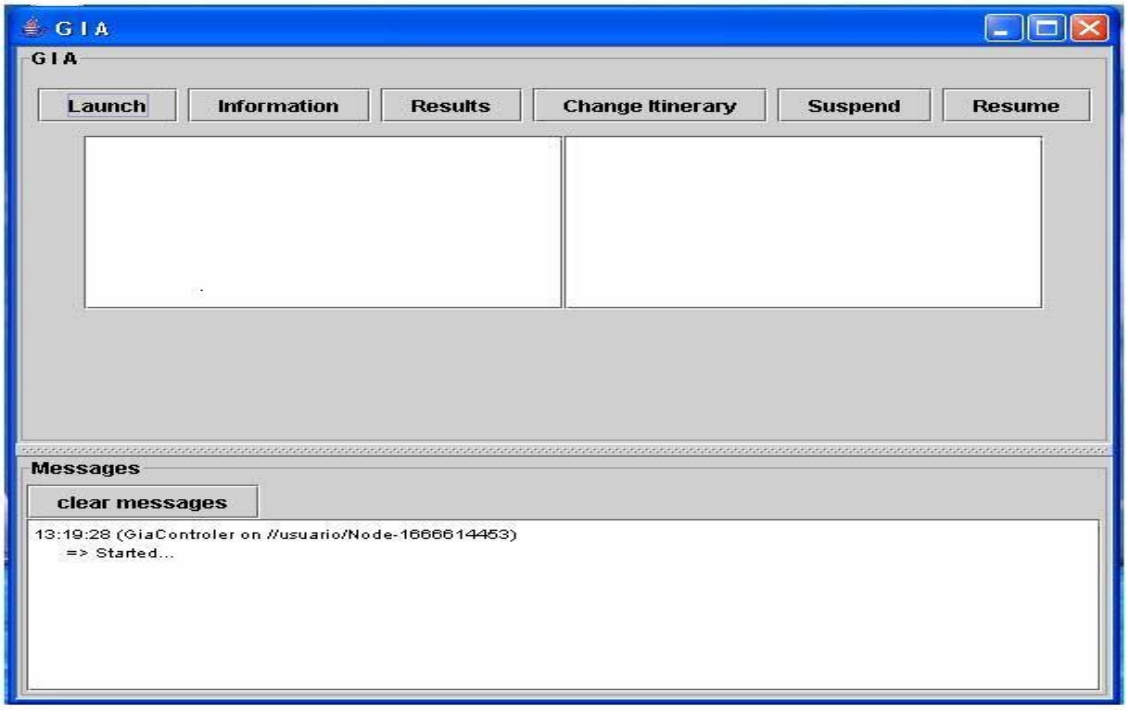

FIGURA 4.31 VENTANA DE LA INTERFAZ DEL GIA

La ventana correspondiente a LaunchAgent, permite introducir el nombre del agente, los parámetros necesarios para ese agente en particular, es posible determinar antes del lanzamiento el itinerario a recorrer por el agente ya sea agregando o removiendo un host. Y con el go se hace un proceso de verificación de la existencia de la clase, si existe la clase se carga en memoria creándose una instancia de la clase, que recorre cada uno de los hosts determinados desde el lanzamiento y que están involucradas en el computo a través de un archivo xml (ver Figura 4.32).

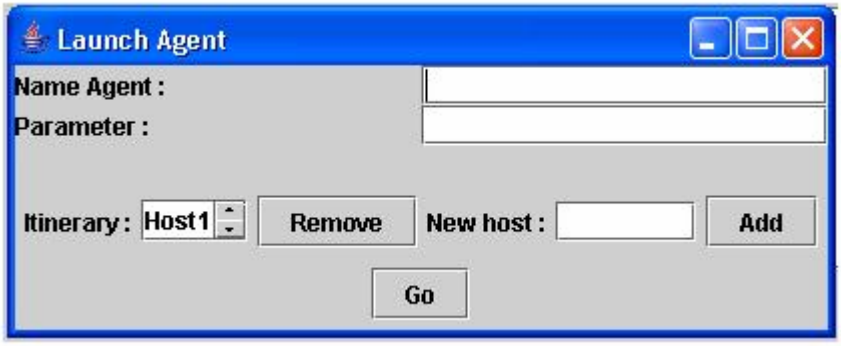

FIGURA 4.32 VENTANA LAUNCHAGENT

En la Figura 4.33 se observa en el área correspondiente a la lista de agentes, dos agentes con un identificador único. Y en el área correspondiente a los mensajes se observa quien esta en actividad.

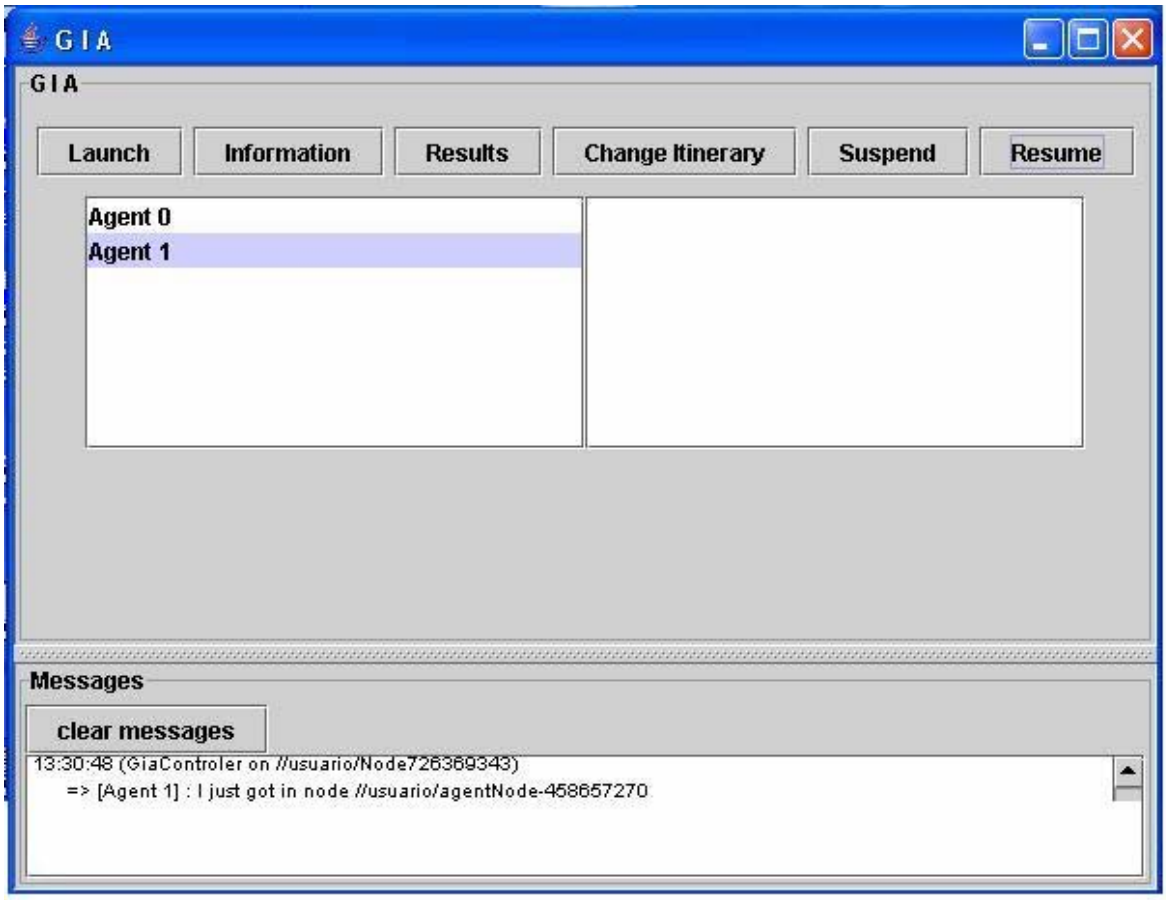

FIGURA 4.33 AGENTES EN ACTIVIDAD

La tabla que se observa en la Figura 4.34, es posible observar, la forma en que la información general de los agentes aparecerá.

FIGURA 4.34 VENTANA INFORMATIONAGENT

La ventana Results of agent, presenta a través de un área de texto el resultado obtenido por el agente una vez finalizada su actividad, con la opción de imprimir la misma (ver Figura 4.35).

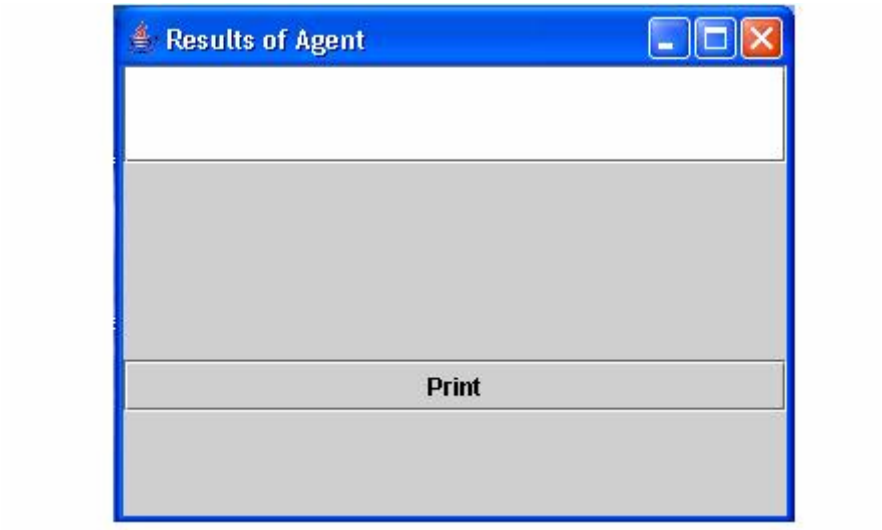

FIGURA 4.35 VENTANA DE RESULTAGENT

En este capítulo se ha presentado el diseño de la infraestructura genérica para agentes móviles así como una visión general de la aplicación de administración de agentes.

Conclusiones

ProActive es una biblioteca que proporciona una infraestructura genérica para el desarrollo de aplicaciones paralelas, distribuidas y concurrentes. ProActive fue desarrollada totalmente en el lenguaje Java y por lo tanto posee capacidad para ambientes multiplataforma. El liderazgo del Dr. Denis Caromel de los laboratorios del INRIA ha sido fundamental para el crecimiento de esta biblioteca.

ProActive PDC es un conjunto de primitivas que permite realizar cómputo paralelo, distribuido y concurrente de forma transparente. El conjunto de primitivas crece día con día. El tratamiento de agentes móviles dentro de ProActive, había sido relegado por un tiempo, debido principalmente a la complejidad de implementación de una infraestructura genérica, sin embargo, en este trabajo de investigación se ha atacado el problema y se ha analizado, diseñado través de un modelado una infraestructura genérica para agentes móviles en ProActive. Tanto el análisis como el diseño fueron realizados haciendo uso del lenguaje de modelado unificado (UML). Es posible observar el cuidado de los detalles en este trabajo, de hecho es posible *realizar* una implementación casi inmediata a partir de los diagramas realizados, trabajo contemplado en un futuro próximo.

Un agente móvil es la consecuencia de la evolución de los objetos estándares, que proporciona características importantes para el desarrollo de aplicaciones distribuidas, debido principalmente a la capacidad de migración.

El hecho de tener la capacidad de generar agentes móviles a través de un ambiente distribuido, incrementa indudablemente la flexibilidad y el rendimiento (entre otras cosas), en el sistema de cómputo distribuido.

La propuesta de este trabajo se fundamenta en la administración de agentes móviles. Para la cual se tiene un prototipo que proporciona una interfaz para realizar tal actividad. La administración involucra actividades como las siguientes:

- Lanzamiento de agentes.
- Solicitud de información de un agente en particular.
- Obtención de resultados de agentes móviles.
- Suspensión y reactivación de la actividad de un agente.
- Cambio de itinerario para los agentes.

La infraestructura diseñada se encuentra soportada por la biblioteca de ProActive, por lo que el uso de objetos futuros se encuentra también disponible en esta propuesta, proporcionando confiabilidad en la comunicación, a través de comunicación asíncrona.

Uno de los objetivos planeados fue proporcionar un estándar en la codificación de los agentes, con la finalidad de permitir la comunicación entre ellos mismos y la interfaz gráfica de agentes (Gia).

El trabajo sienta las bases para el futuro desarrollo de aplicaciones en ambientes distribuidos, paralelos y concurrentes que necesiten del uso de agentes móviles para realizar tareas particulares. Las tareas pueden colaborar entre ellas, ya que los agentes móviles tienen la capacidad de comunicarse entre ellos.

Esta propuesta enriquecerá la biblioteca de ProActive, mediante la incorporación de los nuevos métodos diseñados para el control de agentes móviles.

En este proyecto se usó un ambiente compuesto de equipos heterogéneos para realizar las pruebas de la biblioteca ProActive. El ambiente estuvo compuesto por cuatro equipos, con las siguientes características genéricas: una computadora Intel con sistema operativo Windows XP, una computadora Intel con sistema operativo Linux (RedHat 7), una computadora con procesador SPARC (SUN) con sistema operativo Solaris (versión 9) y una computadora con procesador SPARC (SUN) con sistema operativo Linux (RedHat 7.1).

Las restricciones obtenidas después de la utilización de ProActive en un ambiente heterogéneo son las siguientes:

- Es necesario utilizar la misma versión de Java en cada una de las computadoras.
- Es necesario usar la misma versión de ProActive en cada una de las computadoras.

Como se puede observar, los requerimientos son mínimos, comparados con las bondades obtenidas con el uso de una infraestructura genérica para el cómputo paralelo, distribuido y concurrente.

Finalmente, es importante mencionar que este trabajo ha sido presentado en el sexto Congreso Internacional en Telecomunicaciones e Informática llevado a cabo durante el mes de Mayo del 2004 en la Ciudad de Cancún, México. El trabajo ha sido publicado en las memorias del evento [PHM04a], así como en el "Transactions on Communications" [PHM04b]. De igual manera, el trabajo ha sido aceptado en la 3ra Conferencia Iberoamericana en Sistemas, Cibernética e Informática CISCI 200 que se llevará a cabo en el mes de julio del 2004, en la ciudad de Orlando, Florida, Estados Unidos.

Propuestas para el mejoramiento de este trabajo de tesis:

- Completar la codificación del prototipo Gia.
- Incorporación de autonomía total al Gia para la administración de lo agentes móviles.
- Estudio de mecanismos que permitan adicionar comportamientos avanzados en los agentes, como es el caso de *pause* y *resume*.

Referencias

[BBD91] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville, G. Vandôme. *Architecture and Implementation of Guide, an Object-Oriented Distributed System. Computing Systems*, Vol. 4, nº 1, pp. 31-67, Invierno de 1991.

[CKV98] D.Caromel, W. Klauser, J. Vayssiere. *Towards Seamless Computing and Metacomputing in Java//*. Concurrency Practice and Experience, September-November 1998, 10(11-13), Editor Geoffrey C. Fox, Published by Wiley & Sons, Ltd. bibtex modelo
URL http://www-sop.inria.fr/sloop/javall/

[CRY97] Mobile agent facility specification. Reporte técnico, Crystaliz, Inc.
(EU, Junio 97).
URL http:/www.crystaliz.com

[BOO01] G. Booch. *Object-Oriented Analysis and Design with Applications (2nd Edition),* Addison-Wesley

[BOO99] Gr. Booch et al. *El lenguaje unificado de modelado*, Addison-Wesley, 1999.

[GRA95] R.Gray *Agent Tcl: a transportable agent system*. In proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference 07. Information and Knowledge Management (CIKM95, Baltimore Maryland, Diciembre,1995).
URL http:/www.cs.darmouth.edu/~rgray/transportable.html.

[HBS96] F. Hohl, J. Baumann and M. Straber. *Beyond Java: Merging Corba-based Mobile Agents and www.* Joint W3C/OMG Workshop on Distributed Objects and Mobile Code, June 1996 Boston, Massachusetts.

[JRS95] Johansen D., Renesse,R. y Schneider,F. *An introduction to the TACOMA distributed system.Version 1.0*. Reporte técnico 95-103. Institute of Mathematical and Physical Sciences, Department of Computer Science, University of Tromso, Norway, Junio 1995. URL http://www.cs.uit.no/Lokalt/Rapporter/Reports/9523.html.

[KOK94] K. Kotay, K. Kotz.*Transportable Agents*. Proceedings of the CIKM Workshop on Intelligent Information Agents, Third International Conference on Information and Knowledge Management (CIKM94, Diciembre,1994).

[KTM97] Kato, K., Toumura, K., Matsubara, K. *Ptrotec and secure mobile object computing in Planet*. Reporte técnico. Information Sciences and Electronics, University of Tsukuba, Tusikuba, Japón, 1997.

[KIR97] M. Kirtland. *Object-Oriented Software Development Made Simple with COM+*.

[LEC01] L. Thompson, *Infinite Game Universe:Mathematical Techniques*, Mayo de 2001. Advances in Computer Graphics and Game Development

[LDD95] A. Lingnau, Dronbnik,O., Domel,p.,1995. *An HTTP-based infrastructure for mobile agents*. Proceedings of the 4th International WWW Conference December, 1995.
URL http://www.w3.org/pub/Conference/WVVVV4/Papers/150/

[MIC95] Microsoft Corporation. *The Component Object Model Specification. Microsoft*, Octubre de 1995. URL: http://www.microsoft.com/com.

[MIC98] Microsoft Corporation. Distributed *Component Object Model Protocol. Microsoft*, Enero de 1998. URL: http://www.microsoft.com/com.

[NEW96] H. Nwana, *Software Agents : An overview. intelligent systems research*. Knowledge Engineering Review 11, 3 (Oct/Nov,1996), 205-244. URL http://www.sce.carleton.ca/netmanage/docs/AgentsOverview/ao.html

[OMG99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, revision 2.3. Object Management Group. Junio de 1999.URL: http://www.omg.org.

[OSF92] Open Software Foundation. *Introduction to OSF DCE*. Open Software Foundation, Cambridge, USA, 1992

[PHM04a] D. Pinto, A. Beristain, M. Márquez: *A Generic Infrastructure for Object Streamming and Mobile Agents on ProActive*. On Proceedings of TELEINFO Conference: ISBN: 960-8052-98-X, 2004.

[PHM04b] D. Pinto, A.Beristain, M. Márquez: *A Generic Infrastructure for Object Streamming and Mobile Agents on ProActive*. TELE-INFO'04: Transactions on Communications: ISSN: 1109-2742, Pag. 71-75, 2004.

[SAH97] A. Shauget. *About agents and database*.Reporte técnico CIS-650. Mayo 95.

[SAN96] J. Sanchez, Agents services. Disertacion Doctoral.Department of Computer Science, Texas A&M University, College Station, Texas, Agosto, 1996. URL http://csdl.tamu.edu~joseash/diss.html.

[SAN97] J. Sanchez, *A taxonomy of agents*. Reporte técnico ICT-97-1. ICT. Laboratory of Interactive and Cooperative Technologies, Departament of Computer Systems Engineering,Universidad de las Américas- Puebla, Cholula, Pue.72820, México, Enero, 1997.

[SMM93] B. Schill y Markus U. Mock. DC++: Distributed Object-Oriented System Support on top of OSF DCE. Distributed Systems Engineering, pp. 112-125, 1993.

[SUN98] Sun Microsystems. Java Remote Method Invocation (RMI). URL http://java.sun.com/products/jdk/rmi/index.html

[SZB96] S. Stone, S. Zyda, M..Brutzman, J. Falby. *Mobile agents and smart networks for distributed simulations*. Reporte técnico CSD4. Computer Science Departament, Naval Postgraduate School, Monterey, California, 1996.

[TAV96] J. Tardo, L. Valente. *Mobile Agent Security and Telescri*pt. In IEEE CompCon, 1996.

[TCL99] URL http:// www.uam.edu.ni/uam99/ingieneria/tcl/

[TEL95] URL http:// www.science.gmu.edu/~mchacko/Telescript/docs/telescript.html

[VEN97] B. Venners. *Solve real problems with aglets, a type of mobile age*nt. JavaWorld Under the hood Magazine. Mayo 1997. pp 2-4.

[VIT96] J. Vitek. *Secure object spaces*. Reporte técnico. Object Systems Group, University of Geneva, Geneva.,Switzerland, 1996.

[WHI96] R.White. *Mobile Agents white paper*. Reporte técnico AAA1 Press/the MIT Press., General Magic, Menlo Park, California.

[WOO95] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. In *Knowledge Engineering Review* 10(2), 1995.