INSTITUTO POLITÉCNICO NACIONAL

CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

AGENTES Y COMPONENTES SEGUROS

Reporte de Investigación

Dr. Felipe Rolando Menchaca García[1]
M en C. Jorge Cortés Galicia[2]
M. en C. Gonzalo Pérez Araiza[3]
M. en C. Chadwick Carreto Arellano[4]
Ing. Nora María Flores Fernández[5]

RESUMEN

Este informe presenta las características de distintas arquitecturas de software orientado a objetos y componentes distribuidos existentes en la actualidad y de amplio uso en el desarrollo de aplicaciones de cómputo distribuido. COM/DCOM y JavaBeans son ejemplos de arquitecturas de componentes, mientras que CORBA es el ejemplo fundamental para objetos distribuidos. Se presentan también las principales arquitecturas orientadas a agentes colaborativos.

Se explican los aspectos generales de diseño, poniendo énfasis en cómo están construidas las diferentes arquitecturas y como aseguran el soporte de comunicación entre objetos, componentes o agentes que se encuentran distribuidos a través de la red de comunicación de datos. Se presentan los componentes esenciales de cada arquitectura, la forma de cómo interactúan entre ellos para dar soporte de comunicación y mantenimiento de los distintos elementos, y se revisan aspectos de seguridad de cada arquitectura, así como las vulnerabilidades a su seguridad que éstas han evidenciado.

Palabras clave: Arquitecturas de software, COM/DCOM, CORBA, JavaBeans, Agentes, diseño, seguridad, vulnerabilidad.

[1] Profesor Investigador del Centro de Investigación en Computación del IPN
[2] Alumno del Doctorado en Ciencias de la Computación del Centro de Investigación en Computación del IPN y Profesores de la Escuela Superior de Cómputo del IPN
[3] Alumno del Doctorado en Ciencias de la Computación del Centro de Investigación en Computación del IPN y Profesores de la Escuela Superior de Cómputo del IPN
[4] Alumno del Doctorado en Ciencias de la Computación del Centro de Investigación en Computación del IPN y Profesores de la Escuela Superior de Cómputo del IPN
[5] Docente Investigadora del IPN (ESCOM, CENAC, DCC)

CONTENIDO

Página

1. Introducción

1.1 Definición de arquitectura de software. **1**
1.2 Tipos de arquitecturas de software. **1**
1.3 Aspectos generales de análisis y diseño. **4**
1.4 Aspectos generales de vulnerabilidad. **4**
1.5 Aspectos generales de seguridad. **6**
1.6 Arquitecturas de software actuales. **7**

2. COM-DCOM.

2.1 Modelo de componentes. **10**
2.2 Arquitectura COM. **13**
2.3 Activación, manejo de conexiones y concurrencia. **18**
2.4 Aspectos de relativos de seguridad. **30**
2.5 Aspectos de vulnerabilidad. **32**
2.6 Desarrollo de aplicaciones. **34**

3. JavaBeans

3.1 Introducción. **37**
3.2 Arquitectura de Comunicación de los JavaBeans (INFOBUS). **40**
3.2.1 Tipos de Componentes **41**
3.2.2 Principales requerimientos para el INFOBUS **41**
3.2.3 Protocolo INFOBUS para intercambio de datos **41**
3.3 Seguridad en los JavaBeans. **42**
3.3.1 Características de lenguaje Java **42**
3.3.2 Funcionamiento de Java **44**
3.3.3 Administrador de seguridad **46**
3.4 Vulnerabilidad de los JavaBeans **53**

4. CORBA

4.1 Introducción a la arquitectura CORBA. **55**
4.2 Estructura de un agente de invocación de objetos **55**
4.3 Adaptadores de objetos. **61**
4.4 Repositorio de interfaces **61**
4.5 Protocolo IIOP y GIOP **64**
4.6 Aspectos de seguridad en CORBA **65**
4.7 Aspectos de vulnerabilidad del servicio de seguridad de CORBA **66**

5. Arquitecturas de Agentes

5.1 Introducción

5.2 Elementos en la programación de sistemas de agentes **--**
5.3 Arquitectura FIPA-JADE. **--**
5.4 Arquitectura ADDA **--**

Referencias **--**

1 Unidad I. Introducción

1.1 Definición de Arquitectura de Software

Desde los programas pequeños hasta los grandes sistemas tienen una estructura y un comportamiento que determina su arquitectura. Este concepto de arquitectura, actualmente tiene una gran importancia para el desarrollo del software, porque identifica los elementos que conforman un sistema, su comportamiento e infraestructura de control y de comunicación. "Una arquitectura de software es aquello que comprende la descripción de los elementos a partir de los cuales los sistemas son construidos, las interacciones entre dichos elementos, los patrones que guían su composición y las restricciones de esos patrones [Garlan]".

Otra definición es: "La arquitectura de software de un programa o sistema computacional es la estructura o estructuras del sistema, que comprende a los componentes de software, las propiedades visibles externas de esos componentes y las relaciones entres ellos [vmspec]".

Analizando la definiciones anteriores podemos darnos cuenta que el primer paso es la abstracción del problema para poder construir componentes que cumplan con los objetivos requeridos, posteriormente ver como se logran las interacciones entre los componentes construidos, para así constituir el sistema.

1.2 Tipos de Arquitecturas de Software

Las arquitecturas de software más populares y que por lo tanto han sido también las más relevantes son:

- Arquitectura centralizada
- Arquitectura cliente/servidor
- Arquitecturas orientadas a objetos
- Arquitecturas orientadas a componentes
- Arquitecturas orientadas a agentes

2.1.0 Arquitectura centralizada

En esta arquitectura el sistema informático es único y monolítico; un ejemplo es el sistema informático clásico de una empresa, con el cual soporta y controla la operación de los departamentos de contabilidad y recursos humanos. El sistema está instalado centralmente en un equipo conocido como "host" o "mainframe", al que solo tienen acceso los usuarios autorizados, por medio de una cuenta de usuario.

1.2.2.1 Características

La computadora central ("mainframe") es la única computadora de la empresa.

- Contiene los datos y está dedicada a la consolidación de la información.
- Desde dicha computadora central se controla el acceso a múltiples terminales conectadas a través de productos integrados en la arquitectura de red.
- Las terminales funcionan como "esclavas" de la computadora central.
- Cada usuario tiene una cuenta asignada, así como algunos derechos y prioridades de ejecución en la computadora.

1.2.2.2 Ventajas e Inconvenientes

Entre las principales ventajas de esta arquitectura se encuentran las siguientes:

- Alto rendimiento transaccional.
- Alta disponibilidad.
- Control total de la computadora, al ser única y ubicarse en un único Centro de Cómputo.
- Concentración de todo el personal de explotación y administración del sistema en un único Centro.
- Alto nivel de seguridad.

Entre los **inconvenientes** destacan:

- Es una arquitectura heredada obsoleta, ya sin soporte tecnológico.
- El alto precio de la computadora, al requerirse mucho poder de cómputo para dar servicio a todos los usuarios que estén conectados y gran espacio de almacenamiento para alojar todos los datos de la empresa.
- Alta dependencia de las comunicaciones. En caso de caída de una línea, todos los puestos de trabajo dependientes de dicha línea quedan inoperantes.
- La centralización representa una importante vulnerabilidad en casos de desastre informático.

2.1.1 Arquitectura cliente/servidor

Con la reducción en los costos de las computadoras personales, se dio lugar a que la información de la organización pudiera distribuirse en varias computadoras incluso de tipos diferentes (diferentes plataformas).

El concepto cliente/servidor proporciona una forma eficiente de utilizar los recursos de cómputo de tal forma que la seguridad y la confiabilidad que proporcionan los entornos de cómputo centralizado (“mainframes”) pasan a la responsabilidad de la red de área local. A esto hay que añadir la ventaja del poder y simplicidad de las computadoras personales.

La arquitectura cliente/servidor es un modelo para el desarrollo de sistemas de información, en el que las transacciones se dividen en procesos independientes que cooperan entre sí para intercambiar información, servicios o recursos. Se denomina cliente al proceso que inicia el diálogo y solicita algún recurso o servicio y servidor al proceso que responde a las solicitudes.

En este modelo las aplicaciones se dividen de forma que el servidor controla el recurso que debe ser compartido por varios usuarios, y en el cliente permanece sólo la funcionalidad particular de cada usuario.

Los clientes realizan generalmente funciones como:

- Manejo de la interfaz de usuario.
- Captura y validación de los datos de entrada.
- Generación de consultas e informes sobre las bases de datos.

Por su parte los servidores realizan, entre otras, las siguientes funciones:

- Administración de periféricos compartidos.
- Control de accesos concurrentes a bases de datos compartidas.
- Enlaces de comunicaciones con otras redes de área local o de área extensa.

1.2.2.3 Características

Entre las principales características de la arquitectura cliente/servidor se pueden destacar las siguientes:

- El servidor presenta a todos sus clientes una interfaz única y bien definida.
- El cliente no necesita conocer la lógica del servidor, sólo su interfaz externa.
- El cliente no depende de la ubicación física del servidor, ni del tipo de equipo físico en el que se encuentra, ni de su sistema operativo.
- Los cambios en el servidor implican pocos o ningún cambio en el cliente.

1.2.2.4 Ventajas

- **Aumento de la productividad:** Los usuarios pueden utilizar herramientas que le son familiares, como hojas de cálculo y herramientas de acceso a bases de datos.
- **Menores costos de operación:** El desplazamiento de funciones de la computadora central a servidores o clientes locales origina que los costos del proceso recaigan en computadoras más pequeñas y por tanto, más baratas.
- **Mejora en el rendimiento de la red:** Las arquitecturas cliente/servidor eliminan la necesidad de mover grandes bloques de información por la red, hacia las computadoras personales o estaciones de trabajo, para su proceso. Los servidores controlan los datos, procesan peticiones y después transfieren sólo los datos requeridos a la computadora cliente. Entonces, la computadora cliente presenta los datos al usuario mediante interfaces. Todo esto reduce el tráfico en la red, lo que facilita que pueda soportar un mayor número de usuarios.

1.2.2.5 Inconvenientes

Hay una alta complejidad tecnológica al tener que integrar una diversidad de productos.

- Requiere un fuerte rediseño de todos los elementos involucrados en los sistemas de información (modelos de datos, procesos, interfaces, comunicaciones, almacenamiento de datos, etc.). Además, es un importante problema determinar la mejor forma de dividir las aplicaciones entre la parte cliente y la parte servidor.
- Es más difícil asegurar un elevado grado de seguridad en una red de clientes y servidores que en un sistema con una única computadora centralizada.
- A veces, los problemas de congestión de la red pueden degradar el rendimiento del sistema por abajo de lo que se obtendría con una única computadora (arquitectura centralizada). También la interfaz gráfica de usuario puede volver lento el funcionamiento de la aplicación.

2.1.2 Arquitecturas de software actuales

En la actualidad existen diferentes arquitecturas de software, conocidas y explotadas a nivel comercial, las arquitecturas de software que en el resto del documento se estudian, son aquellas que trabajan con una orientación a objetos y componentes distribuidos. En la nomenclatura inglesa a este tipo de arquitecturas se les conoce como: "*middlewares",* entre las más importantes encontramos a:

- COM/DCOM.
- JavaBeans.
- CORBA.

Cada una de estas arquitecturas presenta características muy propias y que dan como resultado un ambiente de trabajo muy específico; sin embargo también se cuenta con aspectos muy similares entre ellas, sobre todo en su modelo de comunicación. En cuanto a las ventajas de estas arquitecturas, CORBA ha representado una solución a corto plazo para arquitecturas de software distribuidas, aunque DCOM también mantiene un lugar importante y la nueva arquitectura conocida como SOA, empieza a cobrar la relevancia de un estándar de facto.

Actualmente existen productos que soportan el estándar de CORBA 2.3 (por ejemplo Visibroker 4.0, Orbix 2000, MICO, entre otros) y se espera mayor cantidad de productos que soporten el estándar de CORBA 3.0. Además, seguirán evolucionando y ampliándose las funcionalidades de los ORBs *(Object Request Broker*s), mejorando los rendimientos y dando soporte a aspectos tales como seguridad, tolerancia a fallas, manejo de transacciones, etc.

Las ventajas de los monitores transaccionales (p.e. TUXEDO) y los ORBs, se conjugan en los CTM *("Component Transaction Monitor*s") o servidores de aplicación (p.e. Net Dinamycs o MTS - *Microsoft Transaction Serve*r). Estos servidores de aplicación combinan la fluidez y facilidad de acceso de los sistemas de objetos distribuidos basados en un ORB, con un monitor transaccional. Los CTM proporcionan un entorno de componentes, en el servidor gestiona la concurrencia, transacciones, distribución de objetos, balance de carga, seguridad, y gestión de recursos, automáticamente. Esta tecnología, tiende a imponerse, se centra en el desarrollo de programas en Java y PHP.

Existen ya varios CTM que soportan el modelo de *Enterprise JavaBeans* (EJB) y también PHP.

Los Sistemas de Gestión de Bases de Objetos (SGBO) actuales, como ObjectStore o POET, no tienen la madurez suficiente para soportar aplicaciones con grandes volúmenes de datos que requieren rápidos tiempos de respuesta. Para este tipo de aplicaciones se utilizan las nuevas versiones de los productos objeto-relacionales (p.e. Oracle y Universal Server de Informix). Sin embargo, los SGBO pueden ofrecer buenas características para el manejo de Bases de Datos de tamaño pequeño o medio, orientadas a Internet, y sistemas programados en Java.

En cuanto a los lenguajes de programación, se puede pronosticar una mayor utilización de Java y de JavaBeans para el desarrollo basado en componentes de sistemas de gestión. Asistiremos también en los próximos años a una evolución en los lenguajes visuales (p.e Visual Basic), así como a un retroceso del C++ que quedaría más para el desarrollo de software de sistemas y menos para el de sistemas de gestión. Hay que tener en cuenta que este lenguaje no está pensado para un desarrollo basado en componentes. Un avance importante para el desarrollo de aplicaciones distribuidas en la Web es la estrategia de sinergia entre Java y ActiveX basada en OLE/COM, lo que facilita la construcción de componentes COM usando Java del mismo modo que se usaba Visual Basic o C++. Un nuevo e importante actor es PHP que se está convirtiendo en el lenguaje predilecto para el desarrollo de aplicaciones Web.

La Orientación a Objetos (OO) es el punto de partida para lo que se conoce como desarrollo basado en componentes y que no constituye una nueva tecnología, sino una evolución natural de la Orientación a Objetos. La reutilización de componentes ha comenzado con la reutilización de componentes ActiveX, algo que es práctica habitual en la actualidad. Sin embargo, para la reutilización de componentes de negocios queda mucho camino por recorrer. A medio plazo podrán existir componentes funcionales en diferentes dominios de aplicación. Se hace también necesaria la aparición de herramientas que faciliten el desarrollo basado en componentes.

Una evolución de los objetos son los agentes. Los mismos principios de la Orientación a Objetos, lógicamente adaptados y ampliados para este nuevo tipo de objeto, son aplicables al caso de los agentes. Así podemos tener jerarquías de agregación de agentes, colecciones de agentes, patrones de agentes, componentes de agentes, etc. El estándar de OMG permite la movilidad transparente de servicios de objetos a través de una red. Lo que en realidad se puede transferir de un punto a otro de la red son referencias de objetos, pero no objetos completos. La serialización de objetos de Java es una de las aproximaciones más cercanas para solucionar el problema de hacer viajar los objetos a través de la red; sin embargo continúa siendo una solución cara en tiempo de ejecución.

Por otro lado, la Orientación a Objetos está evolucionando hacia la manipulación de objetos físicos distribuidos en una red. La evolución de los dispositivos electrónicos tiende a que cada uno de ellos lleve un procesador o varios que pueden conectarse a una red de datos, brindando así sus servicios. En esta situación será posible que cada dispositivo electrónico (doméstico o industrial) se presente como un objeto distribuido, controlado desde una aplicación central que permitiría que interactuaran entre sí todos estos objetos reales.

1.3 Aspectos generales de análisis y diseño de las arquitecturas de software

El proceso tradicional de diseño propuesto por la Ingeniería de Software, al enfocarse al desarrollo de un sistema y de sus especificaciones, pierde de vista las necesidades comunes y acciones de las personas involucradas en su uso.

El diseño centrado en el humano no tiene formalismos y no es capaz de conectar sistemáticamente las necesidades del usuario y la estructura del software.

El diseño se puede llevar a cabo a través de patrones de software, un patrón es una descripción (forma) que relaciona una solución que resuelve un problema general de diseño (función) con un contexto particular.

Existen varios elementos de representación; es decir aspectos del planteamiento del problema y de la solución. Entre estos podemos mencionar:

- Propósito/objetivo: lo que el cliente quiere
- Forma: lo que el sistema es
- Comportamiento o funcionalidad: lo que hace el sistema
- Desempeño: que tan efectivo es el sistema
- Datos: información del sistema e interrelaciones
- Administrativo: proceso por el cual el sistema es construido y administrado

Las fases del proceso de diseño en una arquitectura de software son: planteamiento del problema, análisis de requerimientos, diseño a nivel de sistema, diseño detallado, integración del sistema, pruebas, sistema generado.

El diseño de la arquitectura de software busca un equilibrio entre el proceso formal para definir especificaciones que deriven en un sistema y el entender el área de trabajo en la cual las personas interactúan con las computadoras.

1.4 Vulnerabilidades.

Para entender los puntos vulnerables que pueden existir en una arquitectura de software, es conveniente entender las etapas involucradas en la creación de la arquitectura, para que a partir de ahí se puedan analizar las posibles vulnerabilidades que cada etapa puede implicar [Redmond]. De manera general, las etapas que están involucradas en la especificación de una arquitectura son:

- Representación.
- Análisis.
  - Estático.
  - Dinámico.
- Diseño
- Evolución
  - En tiempo de especificación.
  - En tiempo de ejecución.
- Refinamiento.

- Seguimiento.
- Simulación/Ejecución.

Cada una de estas etapas puede implicar vulnerabilidades que se deben evitar implementando aspectos de seguridad, los cuales se revisarán en el siguiente apartado.

Comenzando con el análisis de vulnerabilidades, por etapas se tienen los siguientes aspectos:

Representación. La principal vulnerabilidad en esta etapa radica en la posible carencia de un nivel adecuado de comprensión y conocimiento de la finalidad que tendrá la arquitectura; si no se cuenta con una sólida especificación y con el enfoque preciso de la problemática sobre la cual estará inmersa la arquitectura, se presentarán problemas de funcionamiento óptimo de la misma [Pinnock].

Análisis. Dentro del análisis existen dos posibilidades: el análisis estático y el análisis dinámico. Para ambos análisis las principales vulnerabilidades detectadas son una evaluación pobre de los requerimientos globales del sistema que tiendan a reducir al mínimo el número y el costo de errores de interpretación. Por otra parte, se detecta la posible vulnerabilidad de la falta de una idea clara del funcionamiento de la arquitectura a nivel abstracto, ya que no se cuenta con una ejecución real en esta etapa.

Diseño. En el diseño, la principal vulnerabilidad se localiza en la carencia de una especificación adecuada de las partes que integrarán la arquitectura; especialmente en los aspectos relacionados con el tamaño, la distribución y la heterogeneidad de las partes que conformarán dicha arquitectura. Otro aspecto importante es considerar los tipos de sistemas a los que dará soporte la arquitectura de software.

Evolución. Dentro de la evolución existen dos posibilidades: la evolución al tiempo de la especificación y la evolución al tiempo de ejecución. Para ambas evoluciones la principal vulnerabilidad detectada es una falta de visión a futuro para diseñar una serie de elementos, sistemas y familias de sistemas de seguridad capaces de crecer y/o transformarse de acuerdo a la evolución del ambiente sobre el cual opera la arquitectura. Esta visión a futuro debe realizarse en dos niveles, tanto en el momento de la especificación de la arquitectura como en el momento en que se pone en ejecución la misma [Sessions]. Si no existe un adecuado diseño para la evolución de la arquitectura de software en ambos niveles, la vulnerabilidad de la arquitectura será demasiado alta provocando fuertes problemas de seguridad.

Es en esta etapa es donde deben prevenirse posibles ataques a la seguridad de la arquitectura, definiendo una serie de niveles de protección ante ataques, y las estrategias de evolución ante ataques no previstos en la especificación, pero si durante la ejecución.

Refinamiento. La principal vulnerabilidad en esta etapa se produce si no existe un puente de enlace bien definido entre la especificación de la arquitectura y los lenguajes de programación soportados, dejando vacíos de interoperabilidad entre la arquitectura de software y los lenguajes utilizados para explotar las características de dicha arquitectura.

Seguimiento. En el seguimiento encontramos que la vulnerabilidad más probable es la existencia de múltiples perspectivas de la arquitectura debido a la existencia de múltiples niveles de abstracción, lo cual puede provocar múltiples interpretaciones de información cuando se requiere un único enfoque de esta información.

Simulación/Ejecución. Las principales vulnerabilidades que se presentan dentro de esta etapa tienen que ver con la carencia de un modelo de simulación adecuado para probar el prototipo de la arquitectura, si el modelo es inadecuado, es muy probable que se omita la revisión de huecos de seguridad, los cuales sean explotados posteriormente por medio de ataques a dichos huecos. Por otra parte, la falta de verificación de los requerimientos necesarios durante una ejecución dinámica de la arquitectura pueden provocar estados inestables durante la ejecución de la misma.

Estos son los aspectos generales de vulnerabilidad que pueden presentarse dentro de una arquitectura de software, en el apartado siguiente se revisarán los aspectos de seguridad necesarios para prevenir las posibles vulnerabilidades de la arquitectura.

1.5 Aspectos generales de seguridad

En el apartado anterior se revisaron las vulnerabilidades que existen en cada una de las etapas de desarrollo de una arquitectura de software, en este apartado revisaremos los aspectos de seguridad necesarios para prevenir las vulnerabilidades de la arquitectura.

Representación. El nivel de seguridad requerido para prevenir vulnerabilidades en esta etapa radica en el manejo de múltiples perspectivas de la problemática que intenta resolver la arquitectura de software. Estas múltiples perspectivas pueden ser especificarse por medio de notaciones gráficas, técnicas de control de flujo y de control de datos, especificación de procesos a desarrollar, y determinaciones de la utilización de recursos. Una práctica útil es el modelado de configuraciones de la arquitectura en diferentes ambientes sobre los cuales se piensa que ésta interactuará.

Análisis. En esta etapa la seguridad se debe enfocar a los dos distintos tipos de análisis: el estático y el dinámico. Bajo el enfoque estático la seguridad de la etapa del diseño debe enfocarse a la consistencia interna de la arquitectura, a la selección cuidadosa de las propiedades de concurrencia y distribución de elementos, y a diseños heurísticos y con reglas de estilos. Para lograr estos objetivos es necesario contar con analizadores sintácticos, compiladores y verificadores de modelo, así como reglas de distribución y acceso a elementos compartidos.

Bajo el enfoque dinámico la seguridad en esta etapa debe orientarse a la evaluación y depuración de la arquitectura, a la verificación de hipótesis, y a la especificación y verificación de propiedades importantes de la arquitectura a tiempo de ejecución. Para lograr estos propósitos es necesario crear escenarios, descubrir propiedades relevantes a través de simulaciones, y crear filtros de información.

Diseño. En esta etapa la seguridad debe concentrarse en contar con múltiples perspectivas funcionales de la arquitectura y con la generación de guías racionales de diseño y modelado. La manera de alcanzar lo anterior es determinando el tipo de definición de la especificación: pro-activa o reactiva, permitir intrusión o sin intrusión.

Evolución. El nivel de seguridad en esta etapa se centra en dos aspectos relacionados con el tipo de evolución. Tanto en la evolución a tiempo de especificación como a tiempo de ejecución, la seguridad debe concentrarse en crear equivalentes de la arquitectura para refinación, una especificación incremental y modular, y familias de sistemas (módulos) que den soporte a la arquitectura; principalmente soluciones de sistemas de protección contra ataques. Para alcanzar estos objetivos será necesario contar con mecanismos de formulación de prototipos, flexibles y heterogéneos, crear conexiones flexibles y explícitas, y especificar diversas familias de aplicaciones.

Refinamiento. La seguridad en esta etapa debe enfocarse a la creación de diferentes niveles de abstracción para especificar la arquitectura, y a la correcta y consistente refinación de la especificación a través de los niveles de abstracción definidos. Los mecanismos para lograr la seguridad en esta etapa son los mapeos de prevención de inconsistencias y de simulaciones comparativas de los mapeos creados para la arquitectura.

Seguimiento. Dentro de esta etapa la seguridad tendrá que centrarse en el seguimiento cercano y estricto de los diferentes niveles y secciones que integran la arquitectura para detectar puntos vulnerables. Se requiere verificar la manera en que se relaciona cada nivel con los otros y que esa relación esté establecida de forma adecuada, de acuerdo a la perspectiva original de la arquitectura. Además, debe verificarse que los requerimientos de la arquitectura se estén cumpliendo.

Simulación/Ejecución. La seguridad en la simulación/ejecución debe centrarse en la creación de modelos de simulación adecuados y al soporte sistemático para la generación de sistemas orientados a la arquitectura. Para ello deben planearse los diferentes ambientes extremos sobre los cuales la arquitectura debe funcionar y responder adecuadamente. Sobre estos ambientes extremos debe probarse la arquitectura primero a nivel simulación y posteriormente en ejecución.

2 Unidad II. COM/DCOM

2.1 Modelo de Componentes

2.1.1. Introducción

La idea de componente ha aparecido desde los primeros desarrollos modulares de software; las primeras ideas de modularidad surgieron en respuesta al desarrollo de la micro y nanoelectrónica que se observa en el hardware de los equipos de cómputo. La modularidad es bien conocida y de extensa aplicación en la electrónica actual, baste observar un encapsulado de silicio, dentro de él existe una serie de circuitos, los cuales a su vez están compuestos por dispositivos electrónicos más simples que en su conjunto dan una funcionalidad concreta al circuito [Major], cada circuito dentro del encapsulado es un componente con funcionalidades muy específicas, acordes al entorno.

Regresando de nuevo al encapsulado de silicio, éste a su vez puede integrarse a un circuito más complejo (por ejemplo, un sistema mínimo) como un componente más del sistema, proporcionando la funcionalidad requerida. Así pues, no solo se encapsulan funciones específicas en componentes, sino que es posible su reutilización en múltiples oportunidades con el simple costo de adquirir o desarrollar el componente y ajustarlo a los requerimientos.

La idea que el modelo de componentes de software trata de implantar, transfiriendo los conceptos de probada funcionalidad del modelo de componentes de hardware hacia el ambiente lógico de programación. El objetivo es claro tras este modelo, facilitar el desarrollo de aplicaciones modulares, robustas y expandibles de acuerdo a las necesidades de crecimiento.

En el resto del capítulo realizaremos un estudio de este modelo de componentes enfocado al caso particular del modelo de componentes especificado, desarrollado y mantenido por Microsoft. Dicho modelo es el conocido como Modelo de Objetos Componente (COM por sus siglas en inglés) el cual es de extenso uso en todos los productos desarrollados por esta empresa de software, de ahí la importancia de conocer a detalle el funcionamiento de esta tecnología que permite construir dichos productos.

2.1.2. Programación orientada a componentes

Una aplicación generalmente queda construida en un archivo binario, como resultado del proceso de compilación. Después que el compilador genera la aplicación, ésta no puede modificarse y, por lo tanto, no será posible realizarle actualizaciones hasta que no sean trabajadas en alto nivel y la aplicación sea recompilada para generar una nueva versión de la misma.

Esta forma de programar, conocida como estática, restringe en gran medida la idea de manejo de componentes modulares que podría permitir la creación de aplicaciones extensibles. Sin embargo, se ha encontrado la forma de generar aplicaciones dinámicamente, es decir programar aplicaciones de ejecución dinámica. La idea general

detrás de este tipo de programación es dividir las aplicaciones en partes muy básicas con funcionalidades muy específicas y que puedan ofrecer su funcionalidad para otras aplicaciones de mayor tamaño.

A estas pequeñas partes se les puede denominar componentes en un sentido amplio, y concretamente son pequeñas aplicaciones con funcionalidades muy específicas. En un sentido más técnico, son códigos que implantan un algoritmo específico, y están escritos en algún lenguaje de programación, los cuales han sido compilados, enlazados y preparados para ser usados en cualquier momento, por aplicaciones que requieran de la función que ofrecen dichos códigos [Sessions]. Un aspecto importante a resaltar es que un componente por si mismo no permite acceder a su funcionalidad, es necesario crear un contenedor donde el componente pueda ser insertado y utilizado.

Además de las ventajas de la modularidad y la extensibilidad, otras ventajas que proporciona la utilización de componentes son:

- Dos o más componentes pueden ofrecer la misma funcionalidad con diferente algoritmo, esto es, podrán realizar el mismo servicio pero de diferente forma y por lo tanto con diferente eficiencia.
- Rápida creación de aplicaciones, ya que se puede disponer de bibliotecas de componentes. De esta manera, construir aplicaciones complejas consistirá básicamente en reconocer las funcionalidades que requiere, buscar en la biblioteca los componentes adecuados a sus necesidades y agregarlos.
- Posibilidad de acceder a ellos local o remotamente, es decir, se podrán usar los componentes de que se dispongan en la máquina local, así como otros componentes ubicados en otras máquinas con repositorios de componentes instalados, los cuales son accesibles a través de la red.

Para poder beneficiarnos de estas ventajas se han de cumplir principalmente dos requisitos [Grimes]:

Los componentes se deben cargar en la aplicación dinámicamente, es decir, que una vez que la aplicación comienza su ejecución, debe cargar en ese período de ejecución los componentes que va a utilizar. De esta forma, la aplicación usará la versión actual del componente y cuando ésta se actualice, en la próxima ejecución del programa se utilizará la nueva versión del componente, sin necesidad de haber actuado sobre dicho código de programa.

El otro requisito fundamental es el encapsulado; la aplicación que vaya a utilizar un componente, la cual generalmente se denomina aplicación cliente, no tiene porque saber nada acerca de la estructura interna del componente; la aplicación cliente solo conoce la forma de utilizar los servicios que el componente proporciona, sin tener en cuenta el lenguaje de programación en el que está escrito el componente y los detalles de implantación. Un aspecto importante y del que el creador del componente debe estar consciente es que con cada nueva actualización de su componente se deberá mantener la forma en la que las aplicaciones cliente pueden acceder al servicio del componente.

2.1.3 Tecnología OLE

La tecnología OLE representa uno de los primeros intentos para crear un modelo de componentes funcional. Microsoft desarrollando sus primeros sistemas operativos Windows se encontró con un problema: "Necesitaban poder insertar gráficos de una de sus aplicaciones en otra de ellas". En 1991 diseñaron un protocolo mediante el cual en un documento podrían insertarse objetos trabajándose en un programa distinto de aquel en que se estaba editando el documento.

El protocolo se llamó OLE ("Object Linking and Embedding") versión 1.0. Su fundamento técnico está basado en el paso de mensajes y en el uso de memoria global compartida [Pinnock]. El resultado fue realmente malo ya que existían problemas como:

- Vulnerabilidad del sistema respecto a las aplicaciones con uso del protocolo OLE (si caía una de las aplicaciones, el sistema se ubicaba en un estado inestable),
- Alta complejidad en la realización de componentes, para los programadores.

Las condiciones desfavorables de esta tecnología forzaron a Microsoft al replanteamiento de la misma y al surgimiento de una nueva tecnología. La siguiente versión de OLE que se desarrolló (OLE 2.0) estuvo basada en el modelo de componentes COM (Component Object Model), para ello se reescribió desde cero todo el trabajo realizado en la primera versión de OLE. En un principio los participantes eran Microsoft y DEC, pero al final DEC abandonó el proyecto y Microsoft continuó con él. Las nuevas versiones que realizó Microsoft de esta nueva tecnología ampliaron el modelo para poder usar los componentes desde distintas computadoras, manejando un entorno distribuido de componentes, así aparece DCOM (Distributed COM).

En la primera versión, los objetos OLE podían insertarse de dos formas:

- **Incrustación:** el objeto es almacenado en la aplicación destino, pero si el objeto original sufre algún cambio en la aplicación fuente, en la aplicación destino no se reflejará el cambio.
- **Vinculación:** el objeto es almacenado en la aplicación destino y se establece un vínculo con el archivo original de forma que si modificamos el objeto desde la aplicación fuente, los cambios afectarán a la aplicación destino.

En la versión 1.0 de OLE se utilizó la técnica DDE (Dynamic Data Exchange - Intercambio Dinámico de Datos); esta técnica se basaba en mantener vínculos de archivos creados con otras aplicaciones dentro de la aplicación contenedora. Su finalidad era la de reflejar automáticamente cualquier cambio realizado en algún archivo vinculado en la aplicación contenedora.

Sin embargo esta técnica presentaba algunos inconvenientes, entre los problemas que evidenció la técnica DDE estuvieron:

- Dependencia de la estructura de directorios.
- Pérdida de vínculos.
- Un gran número de archivos relacionados a un documento.

Para mejorar estos problemas surge OLE en su versión 1.1, el cual ocupa una técnica de vinculación e incrustación de objetos. Esta técnica permite vincular o incrustar un objeto sin que la aplicación contenedora tenga que conocer el tipo de datos de dicho objeto.

Los problemas que surgieron con esta técnica fueron:

- Cada vez que se manipula el objeto en la aplicación que lo utiliza había que abrir una nueva ventana (correspondiente a la aplicación usada para crear dicho objeto).
- Ausencia de un protocolo de comunicación eficiente para establecer la comunicación entre aplicaciones fuente y aplicaciones contenedoras.

2.2. Arquitectura COM/DCOM

2.2.1 Componentes COM/DCOM

COM es una especificación de software de Microsoft abierta al resto de las compañías del mercado [Redmond] [Sessions]. Esta arquitectura permite la creación de componentes de software para su posterior utilización por otras aplicaciones o, incluso, por otros componentes. Permite además que la reutilización de los componentes no dependa de la compañía o persona que lo construya, del lenguaje de programación en que estén escritos, tanto el componente como la aplicación cliente o de la máquina en la que el componente va a ejecutarse.

Para que todo lo anteriormente citado sea posible, debe existir un estándar que sea seguido por los desarrolladores. La arquitectura COM define un estándar binario (para las plataformas Microsoft Windows 2000/NT/95, Apple Macintosh, UNIX) que hace posible la interoperabilidad entre componentes y aplicaciones. Además, y para que todas las operaciones relativas a componentes sean hechas de la misma forma, COM proporciona una biblioteca (COM Library), en la que están implementadas las operaciones más comunes para el manejo de componentes.

Los componentes COM pueden estar escritos en cualquier lenguaje de programación, se almacenan en archivos ejecutables (EXE) o en bibliotecas de enlace dinámico (DLL) y cumplen los requerimientos que conlleva la arquitectura de componentes:

- el encapsulado, ya que son independientes del lenguaje de programación y pueden ser actualizados sin que el cliente detecte los cambios; y
- la carga dinámica de los mismos, por definición de los archivos EXE y DLL.

Los componentes se hacen accesibles a través de interfaces que están constituidas por un conjunto de funciones que se encuentran implantadas en dicho componente, las cuales pueden ser utilizadas por uno o más clientes. Para una aplicación cliente, la interfaz es la parte del componente que le proporciona la funcionalidad que de éste requiere. Además, cuando el cliente se conecta al componente, la interfaz es el único punto de comunicación, ocultándole todos los detalles de la implantación de sus métodos (funciones miembro).

Para el componente, una interfaz es un conjunto de funciones que debe implementar y exponer de forma que los clientes puedan utilizarla [Sessions]; en concreto, una interfaz es una zona de memoria que contiene un arreglo de apuntadores a esas funciones. La especificación COM se encarga de la citada estructura de memoria, mientras que, como ya se mencionó en apartados anteriores, la implantación de las funciones queda a cargo del programador.

Los componentes COM aparecen en forma de archivos, pudiendo implantarse uno o más componentes en cada archivo. Como ya se mencionó anteriormente, estos archivos no solo pueden ser ejecutables: COM empaqueta componentes en DLLs (Dinamic Link Library), aprovechándose así de las ventajas que estas ofrecen. Y según el tipo de archivo y el espacio de direcciones en que se ejecuten los componentes, se distinguen tres tipos de componentes COM:

- Servidores internos del proceso ("in-proccess") que se encuentran almacenados en DLLs y se comportan como corresponde a una biblioteca de enlace dinámico. Se cargan en el mismo espacio de direcciones en que se está ejecutando la aplicación cliente durante el período de ejecución. Este tipo de componentes presenta la ventaja de su sencillez para el cálculo de direcciones, pero al mismo tiempo presentan una limitación, ya que el componente deberá encontrarse en la misma máquina en la que se ejecute la aplicación.
- Servidores locales ("local proccess"), que se encuentran almacenados en archivos ejecutables y se cargan en diferente espacios de direcciones del que se está ejecutando la aplicación cliente en el período de ejecución, aunque también necesariamente en la misma máquina en la que se va a ejecutar la aplicación cliente. Debido a que se corren en diferentes espacios de direcciones, la aplicación cliente y servidor deberán de tener en común un mecanismo que se encargue del paso de parámetros de un lado a otro, ya que, por ejemplo, la dirección relativa 0x000100 significará diferentes direcciones absolutas en cada una de las particiones.
- Servidores remotos ("remote proccess") que se almacenan en archivos ejecutables y se ejecutan en diferente espacios de direcciones en una máquina cualquiera, es decir, permite que el cliente y el servidor se estén ejecutando en diferentes máquinas conectadas a través de la red. Estos son los servidores de los que se ocupa DCOM (que no es otra cosa que la arquitectura COM con soporte distribuido). Al igual que en el caso de los servidores locales, necesitan de un módulo que se encargue de la comunicación entre la aplicación cliente y servidor.

El módulo de comunicación entre cliente y servidor, requerido en los servidores remotos, se encarga de empaquetar los parámetros para que viajen por la red [Redmond]. Esto es necesario tanto en la parte del cliente como en la del servidor, ya que en las llamadas a las funciones, la aplicación cliente debe empaquetar los parámetros de entrada y en la parte del servidor se deben empaquetar los parámetros de retorno. El módulo citado se carga en el mismo espacio de direcciones que la aplicación cliente y que el servidor, por lo que una DLL puede realizar dicha función. El módulo de comunicación está constituido por dos elementos importantes que reciben el nombre de Proxy y Terminación o Cabo ("Stub"). En la figura 2.1 se muestra la arquitectura COM/DCOM.

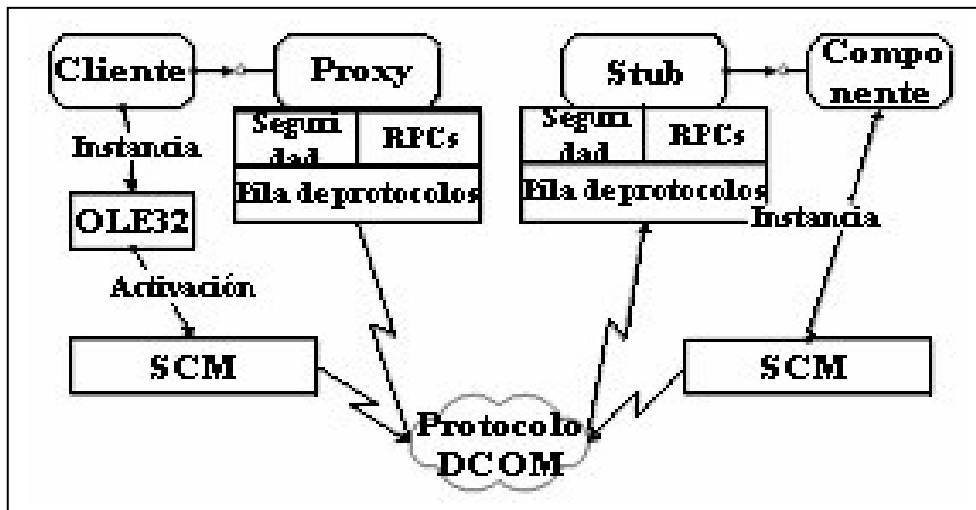


Figura 2.1. Arquitectura general de COM/DCOM

2.2.2 Interfaces

Una vez introducido el término interfaz se pueden distinguir dos características básicas que poseen:

- Las interfaces son invariantes, es decir, una vez implementadas nunca podrán cambiar su estructura. Cuando un componente implementa una interfaz, lo hace dependiendo de las necesidades que se tengan. En la interfaz se pueden implementar una o varias funciones, con uno o varios parámetros en cada función, con tipos determinados. De esta forma, las aplicaciones cliente harán lo necesario para utilizar los métodos que tenga la interfaz, pero siempre se verán condicionadas al número de funciones que tengan o a los parámetros que estas posean. El motivo de que las interfaces deban permanecer invariantes radica en esta condicionalidad de los clientes.
- Si una aplicación cliente está utilizando una determinada interfaz y por cualquier razón, ésta se modifica, ya sea por actualizar la versión del componente, por añadir una nueva funcionalidad a la interfaz o por mejorar el funcionamiento del mismo; si no se mantiene la antigua estructura que la interfaz poseía, habría que modificar la aplicación cliente para que ésta pudiera usar la nueva funcionalidad. Esto no concuerda con la filosofía de componentes, expuesta anteriormente, por lo que para solucionar este problema es necesario que las interfaces sean invariantes.
- Siempre que se quiera modificar la estructura de una interfaz se deberá generar una nueva. Si por el contrario, no se va a modificar la estructura sino que únicamente se va a modificar algún detalle en la implantación (aspectos que no interesan a COM), no será necesario generar una nueva interfaz debido a que el cliente podrá seguir utilizando la misma versión de interfaz sin realizar ningún cambio en su código.
- La segunda característica de las interfaces a destacar, es que permiten al cliente tratar a componentes distintos de la misma manera. COM proporciona una

forma estándar de utilizar interfaces que no depende del componente que los contenga. Esta segunda característica es conocida como polimorfismo. Gracias al polimorfismo, las interfaces no presentan ningún tipo de información relativa a la implantación al cliente, obteniéndose un encapsulado máximo.

2.2.3 Identificación de componentes

Anteriormente se revisaron los diferentes tipos de servidores que puede existir bajo la especificación COM, que no se limita a los que un cliente tenga en su sistema, sino a toda una colección de componentes realizados por otras personas disponibles a través de la red. Esto también ocurre cuando hablamos de los interfaces.

Debido a ello será necesario disponer de una forma de identificar a estas interfaces. Para que la identificación de componentes e interfaces se realice de una forma unívoca, COM proporciona unos identificadores llamados Identificadores Globales Únicos (GUIDs) que son números de 16 bytes de longitud. El algoritmo para la creación de estos números se obtiene a partir de la hora y la fecha de creación del componente y del número de la tarjeta de red del equipo en el que se creó. De esta forma, tanto componentes como interfaces serán fácilmente identificados ya que tendrán una clave que les distinguirá de forma única de todos los existentes. COM supone que con la longitud dada a cada GUID, 2^128, es suficientemente grande para asignar identificadores únicos para todos los componentes e interfaces del mundo.

Para hacer una distinción entre los GUIDs que identifican componentes y los GUIDs que identifican interfaces, a los primeros se les llamara Identificador de Clase (CLSID) mientras que a los segundos se les conoce con el nombre de Identificador de Interfaz (IID).

2.2.4 Ciclo de vida de un componente

A continuación se presenta una breve explicación del funcionamiento completo de una aplicación COM, es decir, de cómo el cliente se conecta con un componente, utiliza sus interfaces y, una vez que ya sabe que no va a volver a necesitar el servidor, se desconecta del componente.

En el primer paso, la aplicación cliente se debe conectar al componente. Esta conexión se realizará llamando a una función específica que se describirá más adelante. En el segundo paso, el sistema realiza una búsqueda del componente que se indicó en el paso anterior. En el punto tercero, lo que se hace es devolver a la aplicación cliente un apuntador al servidor que se encontró. De este modo, en el último paso, el cliente podrá hacer llamadas a los métodos de la interfaz vía el apuntador obtenido.

Falta un último paso que consiste en la desconexión del servidor por parte del cliente. Esto lo hace el cliente, llamando a una función determinada. En esta breve descripción se ha omitido un punto sumamente importante que ha quedado reflejado en la teoría de componentes. La teoría COM requiere la carga dinámica de los componentes, así como la carga del mismo componente una sola vez para atender las peticiones de dos o más clientes. La carga dinámica de componentes se cumple: llegado el momento en el que la aplicación cliente debe utilizar los interfaces de un servidor, ésta llamará a la función apropiada para conectarse al componente. Una vez realizada la conexión, el cliente

obtendrá un apuntador al servidor y utilizará las interfaces del mismo. Cuando ya no requiera los servicios del componente, se desconectará de éste, liberando así todos sus vínculos con el servidor.

Si un componente ya ha sido cargado por un cliente, y aún no ha sido liberado, la petición de conexión por parte de un nuevo cliente no debe generar la carga del mismo componente de nuevo, sino que deberá conectarse al componente ya cargado. Y, por último, pero no menos importante, el componente que recibe una orden de descarga no se debe descargar a no ser que no haya un solo componente conectado. Cuando el cliente pide conectarse, (si eso es posible) éste se conecta al servidor, y no percibirá si se ha conectado a un componente que ya estaba cargado o, por el contrario, es el primer cliente que deseaba utilizarlo, motivando por tanto la carga del componente en memoria. Esto obliga al servidor a mantener la responsabilidad de la descarga, nunca el cliente, ya que no tiene forma de saber si existen otros clientes conectados al servidor. Este mecanismo de descarga es extremadamente sencillo: cuando un cliente realiza una conexión con un servidor, incrementa en uno una variable que todos los servidores tienen y que actúa como un contador de clientes conectados. De forma análoga, cuando un cliente manda una petición de desconexión se decrementa en uno la variable citada, encargándose el propio servidor de desconectarse cuando esta sea cero.

Existen determinadas funciones que un cliente siempre tendrá que hacer. Estas funciones están relacionadas con el ciclo de vida del servidor y parece lógico que deberían pertenecer a una interfaz del servidor, ya que van a manipular una variable de éste. Además de esta cuestión, se da la dificultad de que un cliente únicamente puede mantener un apuntador a un componente, que apuntará a la interfaz solicitada y por medio de él se podrá llamar a los métodos de dicha interfaz. Debido a esta situación se debe crear un mecanismo que permita al cliente decir al componente que quiere cambiar su apuntador. Este mecanismo lo deben usar todos los clientes e involucra tanto a cliente (pidiendo el cambio) como al servidor (ejecutando esa petición y devolviendo un nuevo apuntador).

Parece lógico que tanto estas funciones como el mecanismo anteriormente citado (que todo cliente siempre tendrá que ejecutar) se hayan implantado en una interfaz, de tal forma que todos las interfaces que se construyan puedan heredar estas implantaciones. COM proporciona una forma sencilla de hacer que todos las interfaces posean estos mecanismos: a través de la interfaz IUnknown, que tendrá tres métodos que se encargan de implantar los mecanismos necesarios que se revisaron anteriormente. Absolutamente todas las interfaces que se definan derivan de la interfaz IUnknown, es decir, las funciones que posee IUnknown las heredan las interfaces que se creen (en realidad, no se produce una herencia verdadera ya que los tres métodos que posee la interfaz IUnknown son funciones virtuales puras, esto es, estas no se encuentran implantadas en IUnknown sino que serán implantadas en las interfaces que se creen a partir de ésta). Esto no es un gran problema ya que se puede encontrar la implantación de estas funciones en libros de consulta, URLs de Microsoft e incluso en la ayuda de Visual C++ por lo que no será demasiado costoso copiar estas líneas de código y añadirlas a nuestras interfaces.

Así, la declaración de una interfaz deberá ser como sigue:

```
//Interface IPrimerInt
class IPrimerInt : public IUnknown
{
 //Funciones de IUnknown
 virtual HRESULT QueryInterface(IID iid, void**ppv) = 0;
 virtual ULONG AddRef ( ) = 0;
 virtual ULONG Release ( ) = 0;
 //Funciones propias de la interfaz
 virtual void Funcion1 ( ) = 0;
 virtual void Funcion2 ( ) = 0;
};
```

2.3 Activación, manejo de conexiones y concurrencia

2.3.1 Cliente

Las aplicaciones cliente son las que utilizan los componentes COM/DCOM y pueden estar construidas, en cualquier lenguaje de programación (por lo general, se encuentran en C++, Visual C++ o Visual Basic, aunque no se descarta casi ningún lenguaje, debido a que si no se cumple el requisito anterior, se puede construir una DLL con Visual C++ que la haga de cliente para que así otros lenguajes puedan ejercer como clientes de un componente). El único requisito existente para el lenguaje es que se pueda llamar a una función a través de un apuntador.

Una aplicación cliente realizará tres acciones al momento de manejar componentes: conexión a un componente, uso de los métodos de las interfaces del mismo y, por último, la desconexión del componente. A continuación vamos a ver en detalle como realizan estas acciones.

Para conectarse a un componente, tanto en COM como DCOM, existen varias funciones en la biblioteca COM. El primer paso será conectarse al servidor y concretamente al componente ClassFactory, lo cual se hace mediante la función:

```
HRESULT CoGetClassObject (clsid, grfContext, pServerInfo, iid, ppv);
```

Los argumentos de la función están definidos por:

CLSID clsid = Identificador del componente.
CLSCTX grfContext = Tipo de servidor al que nos conectamos (en el proceso, local o remoto).
COSERVERINFO* pServerInfo = Reservado para DCOM, que indicará el nombre de la máquina donde se encuentra el servidor. Para COM será NULL.
IID iid = Identificador del componente ClassFactory.
void ppv** = Apuntador al componente ClassFactory devuelto. Si la llamada a la función no falla, con esta

variable se podrán llamar a los métodos del interfaz IClassFactory.

La función devolverá un tipo de dato HRESULT, que acepta los siguientes valores:

S_OK =	La llamada ha tenido éxito.
REGDB_E_CLASSNOTREG =	El componente correspondiente al CLSID no puede ser localizado.
E_OUTMEMORY =	No puede ser reservada la memoria para el componente.
E_UNEXPECTED =	Error desconocido.

Esta función se encarga de cargar el servidor en memoria y crear un componente ClassFactory en dicho servidor. Si el servidor ya hubiese sido cargado por otro cliente, una llamada a esta función por otra aplicación no lanzará otro servidor, sino que únicamente se conectará al que ya existe.

Una vez creado el componente ClassFactory falta por crear una instancia del componente que se quiere usar, y esto lo realiza el componente ClassFactory mediante una función que posee su interfaz que es:

```
HRESULT IClassFactory::CreateInstance (pUnkOuter, iid, ppvObject);
```

Donde los argumentos de la función están definidos por:

IUnknown* pUnkOuter =	parámetro usado para la agregación de componentes.
IID iid =	Identificador del interfaz al que nos queremos conectar.
void ppvObject** =	Apuntador devuelto al componente.

La función devuelve un valor HRESUL que acepta los mismos valores que CoGetClassObject.

A partir de este momento el cliente posee en la variable ppvObject un apuntador a la interfaz del componente y, por tanto, podrá utilizar los métodos de esta interfaz, pero antes habrá que decrementar el contador de referencia del componente IClassFactory para que, si no hay ningún otro cliente conectado a él, se desconecte, con lo que finaliza la etapa de conexión con el componente.

Ahora lo que se tiene es un apuntador a la interfaz IUnknown que contiene las funciones QueryInterface, AddRef y Release. Como se puede imaginar una aplicación cliente no querrá llamar a esas funciones directamente, sino a unas funciones específicas que tengan las interfaces del servidor. Para esto, al tener una conexión a la interfaz IUnknown podrá llamar a QueryInterface de la siguiente forma:

```
QueryInterface (IID_IIntPrimero, &ppv);
```

Con lo que si el componente al que estamos conectados soporta la interfaz que especifica la variable IID_IIntPrimero, el apuntador ppv será conectado a dicha interfaz y podrán ser utilizadas las funciones de la interfaz IntPrimero.

No debemos olvidar de realizar un Release sobre IUnknown, que todavía sigue activa. Lo cual no impide que se usen métodos de otra interfaz, pues como ya se explicó, todas las interfaces derivan de IUnknown y deberán implantar (se dispone de ellas en cualquier biblioteca) las funciones que ésta posee, y por tanto, también la función QueryInterface, que nos dará el apuntador a esa nueva interfaz.

Una vez utilizados los métodos de la interfaz deseados, se deberá liberar la misma, llamando a la función Release de esta interfaz. Habiéndose perdido a partir de ese momento todos los vínculos que se poseían con el servidor.

La semántica que se sigue en C++ para liberar una interfaz es:

```
HRESULT hr = ppvObject-> CreateInstance(NULL, IID_IUnknown, &ppv);
hr = ppv-> QueryInterface (IID_IIntPrimero, &ppv);
hr = ppv-> Release( );
```

Anteriormente se ha visto que la secuencia de instrucciones para conectarse a un servidor pasa siempre por la llamada a las funciones CoGetClassObject, CreateInstance y Release. La biblioteca COM proporciona una única instrucción de la API que condensa las tres llamadas anteriores en una sola. Esta función es, respectivamente para COM y para DCOM, la siguiente:

```
HRESULT CoCreateInstance (clsid, pUnkOuter, grfContext, iid, ppvObj);
HRESULT CoCreateInstanceEx (clsid, pUnkOuter, grfContext, pServerInfo,
dwCount, rgMultiQI);
```

Los argumentos de estas funciones están definidos por:

COSERVERINFO* pServerInfo =	Parámetro donde se indica la máquina del servidor.
DWORD dwCount =	Número de elementos de la tabla rgMultiQI.
MULTI_QI* rdMultiQI =	Estructura de apuntadores a interfaces. Esta definida por la biblioteca COM de la siguiente manera:

```
Typedef struct tagMULTI_QI
{
REFIID riid;          // IID de la interfaz
void* pvObj;          // apuntador a la interfaz
HRESULT hr;           // hr devuelto por QueryInterface
}MULTI_QI;
```

Esta estructura, de gran utilidad, ha causado un espectacular cambio en la tecnología COM. Anteriormente, un cliente solo tenía conexión con un componente por medio de una única interfaz, y para usar los métodos de otra interfaz, debía ejecutar QueryInterface y Release, desconectándose del componente actual. Ahora, con esta estructura, se pueden indicar varias interfaces a las que la aplicación cliente se quiere conectar [Pinnock], y por tanto, podrá utilizar sus métodos.

El resto de los parámetros y valores devueltos en el tipo de dato HRESULT coinciden con los que se han revisado cuando se llamaba a las funciones separadamente.

En las figuras 2.2 y 2.3 se muestran dos ejemplos completos de clientes, el primer ejemplo es un cliente DCOM y el segundo ejemplo es un cliente COM.

2.3.2 Servidores COM/DCOM

Un servidor es un módulo de código que se encuentra compilado, enlazado y empaquetado en una biblioteca de enlace dinámico (DLL) o en un archivo ejecutable (EXE). Este archivo implanta uno o varios componentes identificados unívocamente por su CLSID. Los componentes encapsulados en el servidor deben estar integrados de tal forma que las aplicaciones clientes sean capaces de crear y usar esos componentes partiendo de su CLSID.

Un servidor, a su vez, puede ser cliente de otros componentes [Sessions]. Esto generalmente ocurre cuando un servidor se ayuda de otros componentes para implantar parte de sus propios componentes. Este tipo de composición es una alternativa que proporciona la biblioteca COM para resolver la falta de herencia que poseen los componentes, los nombres que reciben las alternativas son: agregación y contención.

La funcionalidad que prácticamente todo componente servidor debe controlar es la siguiente [Grimes]:

- Asignar un CLSID para cada componente implementado y proporcionar al sistema una forma de relacionar un CLSID con el archivo ya sea DLL o EXE, que lo contiene (registro de componentes).
- Implantar un componente ClassFactory con la interfaz IClassFactory para cada CLSID soportado.
- Presentar el componente ClassFactory de tal forma que la biblioteca COM pueda localizarlo después de que se haya producido la carga en memoria del servidor EXE o DLL.
- Proporcionar una forma de bloquear el servidor para evitar su descarga en determinados momentos.

2.3.2.1 Registro de Componentes

La primera función de la que un servidor se debe ocupar es la de asignar un CLSID para cada componente así como un IID para cada interfaz del componente. Ya se ha explicado en la introducción a COM que tanto CLSID como IID son números (llamados GUIDs) de 16 bytes únicos en el mundo. La biblioteca COM proporciona varias

maneras para conseguir estos números, una de las cuales es el empleo de la herramienta UUIDGEN, o también se puede hacer llamando a la función de la API CoCreateGuid().

Una vez que se han obtenido los GUIs necesarios para la creación de un componente otra de las funciones de las que un servidor se debe ocupar es la de proporcionar al sistema una forma de relacionar CLSIDs con los archivos que los implantan (ya que los clientes únicamente trabajan con los CLSIDs e IIDs de los componentes y no aprecian en ningún momento el nombre del archivo que los contiene). Debe existir, por lo tanto, un mecanismo por el cual quede asociado un CLSID a un archivo, ya sea este una DLL o un EXE.

```
void main ( )
{
 HRESULT hr;
 CLSID clsid = (0xE07C1BB3,0xE0DD,0x11D0, 0xBE9C,0x002018288773);
 IID iidInt1 = (0xE07C1BB4,0xE0DD,0x11D0, 0xBE9C,0x002018288773);
 IID iidInt2 = (0xE07C1BB5,0xE0DD,0x11D0, 0xBE9C,0x002018288773);
 IInt1 *pInt1;
 IInt2 *pInt2;
 CString Equipo = OCENTURIONO;
 COSERVERINFO maquina;
 MULTI_QI punteros [2];
 //Llenado del arreglo MULTI_QI
 punteros[0].pIID = &iidInt1;
 punteros[0].pItf = NULL;
 punteros[0].hr = S_OK;
 punteros[1].pIID = &iidInt2;
 punteros[1].pItf = NULL;
 punteros[1].hr = S_OK;
 //Transformamos a UNICODE y copiamos a maquina.pwszName Equipo;
 int nLen = MultiByteToWideChar(CP_ACP, 0, Equipo, -1, NULL, NULL);
 maquina.pwszName = new WCHAR [nLen];
 MultiByteToWideChar(CP_ACP, 0, Equipo, -1, maquina.pwszName, nLen);
 //Instrucción que inicializa la biblioteca COM en forma remota
 hr= CoInitializeEx(NULL, COINIT_APARTMENTTHREADED);
 if (FAILED(hr)) { MessageBox( Error al inicializar la biblioteca ); return; }
 //Conexión con el servidor
 HRESULT hr = ::CoCreateInstanceEx(clsid, NULL,clsctx_remote_server,maquina,2, punteros);
 if (hr == S_OK)
 {
  AfxMessageBox("Conexion establecida con el server\n";
  pInt1=(IInt1*)punteros[0].pItf;
  pInt2=(IInt2*)punteros[1].pItf;
 }
 // Uso de los métodos de las interfaces
 hr =pInt1->PrimerMetodo( );
 if (FAILED(hr)) { MessageBox( Error al llamar a la funcion PrimerMetodo ); return; }
 hr =pInt2->OtroMetodo( );
 if (FAILED(hr))
 {
  MessageBox( Error al llamar a la funcion OtroMetodo );
  return;
 }
 // Liberamos el componente
 hr = pInt1->Release( );
 if (FAILED(hr)) { MessageBox( Error al liberar el componente ); return; }
 hr = pInt2->Release( );
 if (FAILED(hr))  {  MessageBox( Error al liberar el componente ); return; }
}
```

Figura 2.2. Código para crear una aplicación cliente DCOM.

```
void main ( )
{
 HRESULT hr;
 IUnknown *pIUnknown;
 IClassFactory *pClassFactory;
 IInterface1 *ppvObj;
 CLSID clsid = (0xE07C1BB3,0xE0DD,0x11D0, 0xBE9C,0x002018288773);
 IID iid = (0xE07C1BB4,0xE0DD,0x11D0, 0xBE9C,0x002018288773);
 // Instrucción que inicializa la biblioteca COM en forma remota
 hr= CoInitialize(NULL);
 if (FAILED(hr)) { MessageBox( Error al inicializar la librería ); return; }
 // Conexión con el servidor
 hr = ::CoGetClassObject(clsid, CLSCTX_INPROC_SERVER,NULL,IID_IClassFactory,
(void**)&pClassFactory);
 if (FAILED(hr))
 {
  MessageBox( Error en la conexión con el componente );
  return;
 }
 // Creamos la instancia del componente
 hr = pClassFactory->CreateInstance(NUL, IID_IUnknown, (void**)&pIunknown);
 if (FAILED(hr))
 {
  MessageBox( Error al crear la instancia del componente );
  return;
 }
 // Liberar el componente ClassFactory
 hr = pClassFactory->Release( );
 /* Se podría haber hecho la conexión con la interfaz IUnknown de la siguiente forma:
 hr = CoCreateInstance (clsid, NULL, CLSCTX_INPROC_SERVER, iid, (voic*) &ppvObj);
 */
 hr = pIUnknown->QueryInterface(iid, (void**) &ppvObj);
 if (FAILED(hr))
 {
  MessageBox( Error al buscar el interfaz );
  return;
 }
 // Liberamos la interfaz IUnknown
 pIUknown->Release( );
 // Uso de los métodos de las interfaces
 hr =ppvObj->PrimerMetodo( );
 if (FAILED(hr))
 {
  MessageBox( Error al llamar a la función PrimerMetodo );
  return;
 }
 hr =ppvObj->SegundoMetodo( );
 if (FAILED(hr))
 {
  MessageBox( Error al llamar a la función SegundoMetodo );
  return;
 }
 // Liberamos el componente
 hr = ppvObj->Release( );
 if (FAILED(hr))
 {
  MessageBox( Error al liberar el componente );
  return;
 }
```

Figura 2.3. Código para crear una aplicación cliente COM.

Para hacer esto, la biblioteca COM utiliza el Registro de Windows de tal forma que asocia un determinado CLSID con un archivo. Concretamente, utiliza la zona del Registro HKEY_CLASSES_ROOT y la clave CLSID para albergar estos datos [Grimes]. A la hora de registrar los componentes, se producirá una distinción entre el tipo de servidor que se va a registrar. Si el servidor es un servidor dentro del proceso, es decir, que esta implantado en una DLL, se tendrá que registrar el componente con la clave InprocServer32. Si por el contrario, el servidor es un servidor local, la clave del registro será LocalServer32.

Para registrar un componente contenido en un servidor dentro del-proceso, la entrada que se tendría que introducir en el registro sería:

```
CLSID {12345678-ABCD-1234-5678-9ABCDEF00000} = Ejemplo de COM
InprocServer32 = c:\COMS\servidor.dll
```

Mientras que si estuviese implantado en un servidor local sería:

```
CLSID {12345678-ABCD-1234-5678-9ABCDEF00000} = Ejemplo de DCOM
1    LocalServer32 = c:\COMS\servidor.exe
```

Con este sencillo mecanismo, en el Registro de la máquina quedará asociado de una forma permanente un CLSID con el archivo que lo contiene. De este modo, la biblioteca COM siempre podrá localizar la implantación de un componente requerido por una aplicación cliente partiendo de un CLSID.

2.3.2.2 ClassFactory

Un componente ClassFactory es un componente cuya función es crear instancias de un determinado componente. En todo servidor COM/DCOM debe existir un componente de este tipo para cada componente cuya función será la de conectar una aplicación cliente con una instancia del componente.

Para implementar este objeto se debe construir un componente que derive de la interfaz IClassFactory, que es una interfaz definida en la biblioteca COM y que posee las funciones CreateInstance, se encargará de crear una instancia del componente solicitado; y LockServer, que se encargará de bloquear el servidor. Además poseerá, ya que como toda interfaz deriva del IUnknown, las funciones QueryInterface, AddRef y Release. La implantación de este componente será entonces como el mostrado en la figura 2.4.

```
class CClassFactory : public IClassFactory
{
 protected:
 ULONG m_cRef;
 public:
 CClassFactory(void);
 ~CClassFactory(void);
 // Funciones de Iunknown
 HRESULT QueryInterface(REFIID, pLPVOID);
 ULONG AddRef(void);
 ULONG Release(void);
 //Funciones propias de IclassFactory
 HRESULT        CreateInstance(IUnknown*,REFIID        iid,
void**ppv);
 HRESULT LockServer (BOOL);
};
```

Figura 2.4. Implantación del objeto ClassFactory.

La función de este componente es crear instancias de un determinado componente y proporcionar, a la aplicación cliente que lo ordenó, un apuntador a una interfaz de dicho componente. Por regla general, este apuntador suele referenciar a la interfaz IUnknown para que la aplicación cliente a continuación haga una llamada a la función QueryInterface.

La funcionalidad de IClassFactory::CreateInstance (...) es crear un objeto del mismo tipo que el componente solicitado por el cliente e inicializarlo llamando a la función CreateInstance() que todos los componentes deben implantar. Si esta llamada tiene éxito, el componente se habrá creado y lo único que faltará es buscar, llamando a la función QueryInterface, la interfaz que se le ha introducido como parámetro.

La otra función IClassFactory::LockServer(...) se encarga de proporcionar una forma de evitar que el servidor se descargue de memoria en el momento que reciba el último Release de la última aplicación cliente conectada. Esto puede ser útil en componentes que se utilicen prácticamente en todo momento ya que una descarga y luego una posterior carga puede alentar demasiado un determinado proceso. Lógicamente, el emplear esta función exige que se mantenga la conexión con el componente ClassFactory mientras se está utilizando el componente, ya que de lo contrario no se podría llamar a esta función.

2.3.2.3 Descarga de servidores

Estudiado como una aplicación cliente llega a trabajar como un componente deseado, falta por ver como se produce la descarga de memoria de los servidores cuando éstos ya no van a ser utilizados. Primero cabe mencionar que quien tiene que realizar esa descarga es el propio servidor si se trata de servidores locales o remotos (almacenados en archivos EXE), mientras que la aplicación cliente será la encargada de esta descarga

cuando se trate de un servidor dentro del proceso, es decir, almacenado en una biblioteca de enlace dinámico.

La descarga es fácil para el primero de estos casos, pues son procesos independientes y, cuando han comprobado que su contador de referencia y por tanto el número de usuarios conectados a él es cero, se descargará. En cambio, para el segundo caso, las DLLs son módulos de código que son cargados en memoria por las aplicaciones para poder utilizar sus funciones exportables, y cuando la aplicación que la cargó no la va a necesitar más, deberá encargarse de su descarga en memoria. Pero rápidamente aparecen problemas: la aplicación cliente no sabe en ningún momento si existen otros clientes conectados a esa DLL por lo que la descarga del servidor quizás no la pueda hacer cuando acabe de usar los métodos de sus interfaces. Para ello, los clientes utilizan una función de la API de Windows llamada CoFreeUnuSedLibraries( ), que se encarga de descargar todas las bibliotecas que no están siendo utilizadas.

2.3.2.4 Reutilización de componentes

Como se mencionó anteriormente, los componentes COM no aceptan herencia. La herencia que se produce en las clases de C++, donde una clase no implanta ciertas funciones ya que están implantadas en su clase base, no se admite en COM debido a que toda interfaz debe implantar la totalidad de sus métodos. Visto desde otro punto de vista, en COM existe un cierto grado de herencia, ya que, cuando se deriva cualquier interfaz de IUnknown (o de cualquier otra), las funciones QueryInteface, AddRef y Release pasan a formar parte de la nueva interfaz aunque habrá que implantarlas en su totalidad.

Para resolver la posible falta de herencia que poseen los componentes, COM proporciona dos alternativas distintas llamadas contención y agregación que hace que ciertos componentes hereden interfaces de otros ya creados. Para observar estos dos mecanismos se presentarán unos ejemplos que harán que se comprenda con mayor facilidad la reutilización de componentes.

Para explicar la contención, suponga que se desea construir un componente que realice consultas a una base de datos. Este componente poseerá una única interfaz con los métodos que sean necesarios para realizar su cometido. En esta interfaz, deberá haber unas funciones que se encarguen de hacer la conexión a la base de datos, otras que se ocupen de construir la consulta, otras que ejecuten la consulta y por último, otras que se encarguen de recuperar los resultados. Suponga, por otra parte, que en nuestro sistema ya se hizo un componente que tenía una interfaz cuyo cometido era el de hacer la conexión a una base de datos. Sería lógico utilizar esa interfaz para construir el nuevo componente que se esta desarrollando. Ese es el objetivo principal de la contención, aprovechar una interfaz ya construida para construir otra interfaz. El nuevo componente construido utiliza la funcionalidad de una interfaz para implantar sus propias interfaces. Este es el mecanismo conocido como contención.

Para explicar la agregación suponga que en nuestro sistema se desea construir un componente que sea capaz de indexar documentos. Esta indexación podrá ser de diversos tipos y el componente tendrá una interfaz para cada tipo de indexación. Supóngase por otra parte, que en nuestro sistema ya existe un componente que posee una única interfaz cuya función es la de realizar una indexación clásica de un

documento. Será lógico pensar que ese método de indexación no será necesario implantarlo en una interfaz para el nuevo componente que se quiere desarrollar puesto que ya existe una interfaz que hace esa función. En este caso el componente indexador presenta la interfaz que realiza la indexación clásica como si fuese suyo, pero en realidad esa interfaz se encuentra implantada en otro componente. Este es el objetivo principal de la agregación: que un componente presente una interfaz a las aplicaciones de tal forma que no se den cuenta que no lo tiene implantando, sino que se encuentra implantado en otro servidor.

2.3.2.5 Comunicaciones

Después de revisar todo lo referente a servidores, puede existir alguna confusión entre las diferencias existentes entre servidores implantados en DLLs y servidores implantados en archivos EXE. Se observaron unas mínimas diferencias en la implantación de unos y otros relativas a la creación de los componentes ClassFactory, pero estas diferencias no aclaraban la existencia de los dos tipos de servidores.

En toda la explicación de servidores tampoco han aparecido servidores DCOM como un nuevo tipo de servidores sino que se les ha encasillado en el grupo de servidores EXE. No ha aparecido en ningún momento como trata la especificación COM las comunicaciones entre cliente y servidor a través de una red.

Es hora de ver con más detalle las diferencias entre servidores DLL y servidores EXE, además de todo el tema de comunicaciones no tratado hasta ahora.

Todo programa ejecutable genera en nuestro sistema un proceso. Todo proceso tiene asignado por el sistema operativo un espacio de direcciones de memoria que es completamente independiente al de cualquier otro proceso. Las direcciones de memoria que maneja un proceso no son direcciones de memoria lógica. Será el propio sistema operativo el que se encargue de transformar la dirección de memoria lógica que le pase el proceso a una dirección de memoria física que realmente exista.

Mientras que los programas EXE generan un proceso con su propio espacio de memoria, las DLLs son cargadas en el mismo espacio de direcciones de memoria del programa que las utiliza. He aquí la gran diferencia entre los servidores almacenados en DLLs y los servidores EXE. Por esta razón los servidores que están implantados en DLLs se les llama servidores dentro del proceso, ya que se cargan en la memoria de un determinado proceso que en este caso será la aplicación cliente. A los servidores almacenados en archivos ejecutables, siguiendo esta notación, se les llama servidores fuera del proceso, ya que no se cargan en la memoria de la aplicación cliente sino que ellos mismos poseen un espacio propio de direcciones de memoria. En este tipo de servidores se encuentran los servidores locales y los servidores remotos que son servidores fuera del proceso y que se pueden encontrar en una máquina distinta de la aplicación cliente.

Esto supone un grave problema en la utilización de servidores fuera del proceso, no así cuando se emplean servidores dentro del proceso. Se ha visto que una interfaz es una estructura que almacena apuntadores a los métodos del mismo, y que un cliente puede llamar a éstos gracias a que puede acceder a esas direcciones de memoria que hacen

referencia a las funciones de la interfaz. Sin embargo, la aplicación cliente, cuando el servidor se encuentra en un EXE no podrá acceder a la interfaz del servidor, ya que es una estructura que se encuentra en un espacio de memoria que se localiza en un espacio de memoria completamente independiente al suyo. De esta manera, si el cliente no puede acceder a la interfaz, no podrá llamar a los métodos de la misma por lo que la interfaz resultará completamente inútil. Aparecen entonces nuevas necesidades con los servidores EXE que se deberán cumplir:

- Deberá existir una forma de poder llamar a una función aun cuando se encuentre en otro proceso.
- Un proceso deberá ser capaz de pasar datos a otro proceso.
- La aplicación cliente no se debe dar cuenta si esta utilizando un servidor dentro del proceso o un servidor fuera del proceso.

Para que dos procesos puedan comunicarse existen diversas formas, COM utiliza llamadas a procedimientos locales (LPC) que son una forma de comunicar procesos que se encuentran en la misma máquina y están basados en llamadas a procedimientos remotos (RPC), las cuales permiten comunicar aplicaciones que se encuentran en máquinas diferentes. Esta es la forma de comunicación que utiliza COM. La ventaja que tienen las LPC respecto a otros mecanismos de comunicación es que éstas se encuentran implantadas en el propio sistema operativo y, por tanto, éste estará capacitado para llamar a cualquier función de cualquier proceso.

Visto como soluciona COM el problema de las llamadas a funciones de otros procesos, falta por revisar la forma en que van a intercambiar datos los procesos. Se necesitará una forma de intercambiar datos para que tanto cliente como servidor puedan pasarse los parámetros en las llamadas a las funciones, a esto se le conoce como aplanado de datos ("marshaling"). Cuando los dos procesos se encuentren en la misma máquina, es decir, el cliente y el servidor están en el mismo equipo, no parece demasiado complicado. Únicamente habrá que copiar los datos de un espacio de direcciones al espacio de direcciones de memoria del otro proceso. Si por el contrario, los procesos se encuentran en máquinas diferentes, los datos deberán ser empaquetados en un formato estándar debido a las posibles diferencias que puede haber entre dos máquinas. Todos estos procedimientos se realizan en COM por medio de una interfaz llamada Imarshal, y esta tarea la realiza el componente Proxy/stub

2.3.2.6 Proxy/Stub

Uno de los principales objetivos que se definieron al principio de la creación de COM era que la conexión por parte de un cliente con un componente, no se realizara de una forma diferente dependiendo del tipo de servidor (dentro del proceso, local o remoto). Lógicamente la aplicación cliente no debe preocuparse de las LPC o del empaquetado de parámetros aunque, como se vio anteriormente, es completamente necesario.

Para solucionar este problema, el estándar COM indica que se debe de construir un componente dentro del proceso que se encargue de hacer el empaquetado de parámetros y las llamadas a las LPC necesarias. Como es de esperarse este componente deberá integrarse dentro del proceso para que la aplicación cliente y dicho componente compartan memoria y puedan intercambiar datos entre ellos. Esta DLL que trabajará para el cliente se conoce en la terminología COM como proxy.

Evidentemente, el componente también deberá tener una DLL que se encargue de hacer todas las tareas que se citaron anteriormente. En la terminología COM, a este componente se le denomina stub el cual se encargará de hacer las LPC y de empaquetar cualquier parámetro que devuelva el servidor al cliente.

Construcción del componente proxy/stub: para conseguir el componente proxy/stub se necesita compilar y enlazar todos los archivos generados por el compilador MIDL. El MIDL genera cuatro archivos escritos en C que implantan el código del componente proxy/stub de las interfaces que se definieron en un archivo IDL. Para la construcción del componente proxy/stub habrá que compilar los cuatro archivos en una DLL, sin olvidar crear el archivo de definición DEF que todas las DLL deben llevar. Un ejemplo de este archivo sería el mostrado en la figura 2.5.

```
LIBRARY Proxy.dll
DESCRIPTION Proxy/Stub del componente A
EXPORTS
1.1    DllGetClassObject @1
DllCanUnloadNow @2
DllRegisterServer @3
DllUnregisterServer @4
```

Figura 2.5. Archivo de definición de la DLL que implanta el proxy/stub.

Incluyendo este archivo en un proyecto junto con los cuatro archivos generados por MIDL y enlazándolos como una biblioteca de enlace dinámico, tendremos como resultado una DLL que será un componente dentro del proceso que servirá como el proxy para el cliente y el stub para el servidor. Para que al momento de compilar no aparezca ningún error habrá que configurar al compilador para que tome la biblioteca externa rpcrt4.lib y definir la etiqueta REGISTER_PROXY_DLL para indicar que es un componente autorregistrable (ya que todo componente COM debe ir registrado en el Registro de Windows para que pueda ser utilizado).

Con esto se habrá finalizado la construcción del componente proxy/stub, que como se ha podido ver no resulta complicado si se utiliza el lenguaje IDL.

2.3.2.7 Lenguaje IDL

Después de revisar todo lo necesario para que una aplicación cliente sea capaz de comunicarse con un componente fuera del proceso, veamos como construir componentes EXE. Si se parte desde un principio hacia el desarrollo del mismo, se observará que la implantación del componente propiamente dicha no llevaría demasiado tiempo. Pero cuando se llegue a la construcción del componente proxy y del componente stub para el cliente y el servidor, se notará que es necesario tener un amplio conocimiento del lenguaje utilizado para implantar LPCs y para realizar el empaquetamiento de parámetros.

Afortunadamente, COM proporciona una manera sencilla para construir los componentes proxy/stub que reducen de una forma significativa el tiempo de implantación. Para ello bastará con escribir una descripción de las interfaces que nuestro componente implantará en un lenguaje específico llamado IDL ("Interface Definition Language") [3]. A continuación, esta descripción se compilará con la herramienta MIDL ("Microsoft Interface Definition Language Compiler"), que viene integrada en Visual C++, generando los componentes proxy y stub automáticamente. Lógicamente, COM proporciona esta herramienta para ahorrar todo el costoso trabajo que conllevaría implantar dichos componentes, sin embargo, si un programador desea implantar el componente proxy/stub él mismo, puede realizarlo debido a que COM no obliga a utilizar esta herramienta.

2.4 Aspectos de seguridad

2.4.1 Seguridad en COM/DCOM

La seguridad en la tecnología COM/DCOM es un aspecto que no se ha dejado a un lado. Las funciones de la API que se encargan de la seguridad dependen claramente de la plataforma que se utilice. La tecnología COM/DCOM para la plataforma Windows soporta la seguridad que proporciona Windows NT, Novell Netware y DCE Kerberos. La tecnología COM/DCOM proporciona seguridad a todas sus aplicaciones incluyendo las que se ejecutan de manera remota. Existe un nivel de seguridad por omisión que se aplica a todos los programas COM/DCOM existentes de la misma forma que se hace con las aplicaciones OLE. Este nivel de seguridad asegura que la integridad del sistema se va a mantener y principalmente se encarga de evitar que distintos usuarios se conecten a una misma instancia de un componente evitándose así que unos usuarios obtengan datos de otros.

Esta seguridad por omisión puede ampliarse utilizando los mecanismos que la tecnología COM/DCOM proporciona con relación a la seguridad. Estos mecanismos se pueden dividir en dos categorías:

- La llamada seguridad en la conexión que dice como serán creados los nuevos componentes, como se conectarán a las aplicaciones clientes y el tipo de acceso que tendrán a los datos,
- la segunda categoría se llama seguridad en las llamadas y se ocupa de la seguridad en las invocaciones que se puedan producir a lo largo de la red.

Por otra parte, la seguridad referente al transporte de información vía las LPCs o las RPCs están solventadas por el estándar RPC que utiliza la tecnología COM/DCOM, es decir, DCE RPCs.

2.4.2 Funcionamiento de los RPC

Los RPCs representan una poderosa herramienta que permite a los usuarios trabajar en un ambiente de programación cliente/servidor. Es posible a través de los RPC que un cliente pueda invocar directamente a un procedimiento alojado en un servidor remoto. El cliente y el servidor tienen cada uno sus propios espacios de direcciones; es decir,

cada uno tiene sus propios recursos de memoria asignados a los datos usados por el procedimiento.

Cuando un cliente invoca un RPC, la aplicación cliente invoca a un procedimiento local conocido como cabo o terminación (“stub”) en lugar de la implantación del código real del procedimiento al cual desea acceder, en otras palabras, existe un redireccionamiento, vía el cabo, entre la invocación del procedimiento remoto al cual desea tener acceso el cliente y el código que contiene la funcionalidad del procedimiento. Los cabos son generados a través de un compilador IDL para el RPC, y son compilados y enlazados junto con la aplicación cliente. Las funciones que realiza el cabo son las siguientes:

- Recupera los parámetros requeridos del espacio de direcciones del cliente.
- Traduce los parámetros según sea necesario en un formato estándar de NDR para la transmisión en la red.
- Invoca las funciones de la biblioteca RPC del cliente para enviar la petición y sus parámetros al servidor.

El servidor realiza los pasos siguientes para atender el procedimiento remoto:

1. Las funciones de la biblioteca RPC del servidor aceptan la petición y llaman a un código conocido como esqueleto para acceder al procedimiento implantado en el servidor.
2. El esqueleto en el servidor recupera los parámetros del búfer intermedio de la red y los convierte del formato de transmisión en la red (NDR) al formato que el servidor necesita.
3. El esqueleto en el servidor llama el procedimiento real en el servidor.

Entonces es ejecutado el procedimiento remoto dentro del servidor, posiblemente generando parámetros de salida y un valor de retorno. Cuando el procedimiento remoto ha terminado de ejecutarse, una secuencia similar de pasos devuelve al cliente los datos resultantes.

1. El procedimiento remoto devuelve sus datos al esqueleto en el servidor.
2. El esqueleto en el servidor convierte los parámetros de salida al formato requerido para la transmisión sobre la red y los devuelve a las funciones de la biblioteca RPC.
3. Las funciones de la biblioteca RPC del servidor transmiten los datos sobre la red a la computadora del cliente.

El cliente termina el proceso aceptando los datos de la red y devolviéndolos a la función que ha invocado al procedimiento remoto.

1. La biblioteca RPC del cliente recibe los resultados del procedimiento remoto y los devuelve al cabo en el cliente.
2. El cabo en el cliente convierte los datos de su representación NDR, al formato usado por la computadora del cliente. El cabo escribe los datos en la memoria del cliente y devuelve el resultado al programa cliente que invocó.
3. El procedimiento que llama, continúa como si el procedimiento hubiese sido invocado en la misma computadora.

La aplicación del servidor contiene llamadas a las funciones de biblioteca dinámicas del servidor que registran la interfaz del servidor y permiten que el servidor acepte llamadas de procedimiento remoto. La aplicación del servidor también contiene la implantación de los procedimientos remotos que son llamados por las aplicaciones cliente.

2.5 Aspectos de vulnerabilidad

2.5.1 Vulnerabilidades de los RPC DCOM

La vulnerabilidad que presentan los RPC DCOM existe en el subsistema de la arquitectura COM/DCOM, que es un servicio crítico usado por muchas aplicaciones de Windows. El potencial para la explotación remota de aplicaciones se presenta por el hecho de la existencia de COM distribuido (DCOM), el cual permite que los objetos de COM se comuniquen el uno con el otro a través de una red. Sin embargo, el potencial distribuido que presenta COM a través de DCOM trae como consecuencia una fuerte vulnerabilidad de seguridad, ya que DCOM permite que un atacante alcance la vulnerabilidad en COM sobre la red, usando cualesquiera de los puertos siguientes:

- Puerto 135 (“Remote Procedure Call”) de TCP y de UDP.
- TCP vía los puertos 139 y 445 (NetBIOS).
- Puerto 593 (RPC-SOBRE-HTTP) de TCP.
- Cualquier puerto de IIS HTTP/HTTPS si los servicios de Internet COM están habilitados, esta configuración permitiría a interfaces vulnerables COM puedan ser alcanzables mediante cualquier puerto en el cual el servicio Web de IIS esté funcionando.

Los administradores de máquinas afectadas por estas vulnerabilidades deben instalar el Hotfix proporcionado por Microsoft. Además, deberán ser inhabilitados los puertos enumerados anteriormente y los servicios de Internet de COM/DCOM (si está permitido) para prevenir la explotación de esta vulnerabilidad, sin embargo debe tenerse en mente que es posible perder ciertos servicios de Windows, por lo cual se debe de tener cuidado en esta inhabilitación. Los puertos 135 y 593 se pueden filtrar con una posibilidad muy pequeña de interrumpir los servicios en uso sobre la red, sin embargo restringir el tráfico de Intranet en los puertos 139 y 445 se debe hacer cautelosamente.

Como se ha mencionado anteriormente las llamadas a procedimientos remotos (RPC) constituyen un protocolo usado por el sistema operativo de Windows. El RPC proporciona un mecanismo de comunicación entre procesos que permite a un programa que funciona en una computadora ejecutar código en un sistema remoto. Este protocolo se deriva del protocolo del RPC de la fundación del software abierto (OSF), pero con la adición de algunas extensiones específicas de Microsoft.

Hay una vulnerabilidad en la parte del RPC que se ocupa de intercambio del mensaje sobre TCP/IP. La vulnerabilidad resulta debido al manejo incorrecto de mensajes malformados. Esta vulnerabilidad particularmente afecta a la interfaz DCOM con el RPC, la cual escucha en puertos permitidos de RPC. Esta interfaz maneja las peticiones de activación del objeto de DCOM que son enviadas por las máquinas del cliente al servidor. Un atacante que explote con éxito esta vulnerabilidad podría hacer funcionar código con privilegios locales del sistema en un sistema afectado. El atacante podría

realizar cualquier acción en el sistema violado, incluyendo instalación de programas, monitorear, cambiar o suprimir datos, crear nuevas cuentas con privilegios completos, entre otras.

Para explotar esta vulnerabilidad, un atacante necesitaría enviar una petición especialmente formada a la computadora remota, en puertos específicos de RPC. Algunos factores para disminuir estas vulnerabilidades se han citado anteriormente, recordándolas un poco serían las siguientes:

- Para explotar esta vulnerabilidad, el atacante requeriría la capacidad de enviar una petición en los puertos 135, 139, 445 o 593 o a cualesquiera otros puertos específicamente configurados de RPC en la máquina remota. Para ambientes de Intranet, estos puertos estarán normalmente accesibles, pero para Internet, estos puertos deberán ser bloqueados normalmente por un cortafuego (firewall). En el caso donde estos puertos no sean bloqueados, o en una configuración de Intranet, el atacante no requerirá ningún privilegio adicional para explotar la vulnerabilidad.
- Las mejores prácticas de seguridad recomiendan bloquear todos los puertos de TCP/IP que no se estén utilizando realmente, y la mayoría de los cortafuegos incluyendo el cortafuego de la conexión de Internet de Windows (ICF) bloquean esos puertos por defecto. Por esta razón, la mayoría de las máquinas conectadas a Internet deben tener bloqueado RPC sobre el TCP o el UDP. RPC sobre UDP o TCP no es recomendado para ser utilizado en ambientes hostiles tales como Internet. Protocolos más robustos tales como RPC sobre HTTP se proporcionan para estos ambientes hostiles.

2.5.2 Desbordamiento del heap en el servicio de RPCSS de Windows en funciones para la activación de DCOM

Esta vulnerabilidad de la corrupción del "heap" del servicio RPCSS, fue detectada en septiembre del 2003, su severidad es elevada ya que a través de ésta es posible la ejecución remota de código hostil, los sistemas operativos afectados son todos los de la familia de Windows, a excepción de Windows 95, 98 y Me. La vulnerabilidad radica en la forma en que Windows maneja ciertas peticiones remotas vía RPC.

Como se ha mencionado el protocolo de RPC proporciona un mecanismo de comunicación entre procesos permitiendo a un programa que funciona en una computadora local que pueda ejecutar código en un sistema remoto. La vulnerabilidad existe dentro de la interfaz de RPC que maneja DCOM. Esta interfaz maneja las peticiones de activación del objeto de DCOM enviadas por las máquinas del cliente al servidor. La vulnerabilidad detectada se activa enviando un paquete de petición de activación de objetos DCOM malformado con lo cual es posible sobreescribir varias estructuras del "heap" en el servicio RPCSS y permitir la ejecución de código arbitrario.

La vulnerabilidad se puede obtener con un paquete de petición de DCE RPC, seguido por un paquete de petición de activación de un objeto de DCE RPC DCOM malformado. La función CoGetInstanceFromFile del API de COM/DCOM puede generar la petición requerida. Manipulando los campos de la longitud dentro del paquete de la activación, las porciones de la memoria del heap se pueden sobreescribir con los datos que pueden ser definidos por el usuario. El enviar entre 4 y 5 paquetes de la

activación del objeto es generalmente suficiente para disparar la sobreescritura, la cual es manejada vía una excepción. Enviando esta secuencia de paquetes se puede causar continuamente esta excepción dentro del RtlAllocateHeap, que es donde generalmente se atiende esta clase de excepción. Parte del código que se observa al producirse la excepción por desbordamiento del heap es:

PAGE:77FC8F11 mov [ecx], eax
PAGE:77FC8F13 mov [eax+4], ecx

Controlando los valores del eax y del ecx de los registros es posible escribir un dword arbitrario a cualquier dirección de nuestra conveniencia. La ejecución de código hostil se puede realizar a través de diferentes medios, ya sea a través del UnhandledExceptionFilter o un apuntador PEB por ejemplo. Para esta vulnerabilidad específica la mejor ruta sería sobreescribir un apuntador dentro de la sección escribible de datos de RPCSS.DLL: Manipulando los apuntadores de referencia de RtlAllocateHeap y RtlFreeHeap es posible controlar el procesador directamente después de que el heap se sobreescriba.

2.5.3 Desbordamiento del heap en el servicio de RPCSS de Windows en la petición de nombre de archivos en DCOM

Esta vulnerabilidad nuevamente radica en el desbordamiento del heap en el servicio de RPCSS. Específicamente existe un desbordamiento en el búfer que utiliza la interfaz DCOM para el manejo del RPC. La arquitectura de DCOM no realiza ninguna verificación de la longitud del nombre de archivos cuando maneja este parámetro. Los atacantes pueden entonces pasar un nombre de archivo lo suficientemente grande (varios cientos de bytes) como parámetro y causar un desbordamiento del heap provocando un error en el servicio de RPCSS. El tratamiento adecuado de esta excepción entonces puede hacer funcionar código arbitrario en el sistema con privilegio local del sistema. Nuevamente los atacantes pueden tomar cualquier acción en el sistema, instalación de programas, obtención y/o borrado de datos, creando nuevas cuentas con privilegio total, entre otras. Esta vulnerabilidad se puede explotar a través de los puertos 135(TCP/UDP), 139, 445, 593 y otros puertos.

2.6 Desarrollo de aplicaciones

2.6.1 Construcción real de un componente

Cuando en las funciones de conexión con el componente se indicaba que se debía definir una variable del tipo IClassFactory (concretamente un apuntador) que se colocaría como parámetro en la función y tomaría la conexión con el componente ClassFactory, no parecía nada extraño [Grimes]. Lógicamente, el tipo IClassFactory, al igual que el tipo IUnknown, son interfaces que proporciona la biblioteca COM y cualquier aplicación las puede usar definiéndose una variable de ese tipo.

Una vez establecida la conexión con la interfaz IUnknown, se hacía la llamada a la función QueryInterface introduciendo como parámetro una variable del mismo tipo que la interfaz a la que se deseaba acceder. Esto en un principio no parece muy extraño ya que es el mismo caso que para la interfaz IClassFactory o IUnknown. El problema

aparece cuando la aplicación cliente intenta definir esa variable. Suponiendo que la interfaz a la que se quiere conectar está definida en el componente como INumero, cuando el cliente define la variable de la forma típica (Numero *pINumero) se produce un error al compilar ya que esa interfaz la ha implantado un determinado componente [Pinnock], el cual no se encuentra definido en la biblioteca COM como era el caso de IClassFactory o IUnknown.

Otro problema que se evitó en el punto de aplicaciones clientes fue el de los identificadores de componentes GUIDs. En ese punto se daba por hecho que la aplicación cliente sabía los GUIDs que necesitaba, pero el cliente tendrá que obtener en alguna parte de su implantación el valor del CLSID del componente al que se quiere conectar así como los IIDs de las interfaces.

Estos dos aspectos que no se trataron anteriormente tienen una importancia vital al momento de construir aplicaciones clientes, hasta tal punto que si no se solucionan estos problemas, un cliente no se podrá conectar al componente que desee. Para solucionar esto, dependiendo del tipo de servidor al que se desee conectar se actuará de una forma o de otra.

Para el caso de los servidores dentro del proceso, la solución es relativamente sencilla. La cuestión de los GUIDs se soluciona incluyendo en el cliente un archivo que defina los CLSIDs que va a utilizar así como los IIDs de las interfaces. Este archivo es necesario también en el componente, por lo que lo único que habrá que hacer será incluir ese archivo en el código del cliente para que así tenga definidos todos los GUIDs que va a usar de la misma forma que los tiene el componente. El aspecto de las interfaces se soluciona de la misma forma.

Falta por ver como se solucionan estos dos problemas en los servidores fuera del proceso. La forma de solventar estos problemas, se basa en el mismo método que los servidores dentro del proceso, es decir, incluyendo los archivos necesarios, pero cambia la procedencia de esos archivos. Para la cuestión de los GUIDs, se podría hacer de la misma forma que en el caso anterior pero se soluciona de una forma más elegante. Cuando se compila el archivo IDL con la definición de las interfaces se generan cuatro archivos. La aplicación cliente deberá incluir estos archivos así como el CLSID del componente que se desea utilizar ya que ese CLSID no aparece por ningún sitio en el archivo IDL.

Para el aspecto del tipo de las interfaces, la solución es un poco más compleja que en los servidores dentro del proceso. En este caso no bastaría con incluir el archivo del servidor que contuviese la definición de la interfaz ya que en este tipo de servidores es necesario el uso de LPCs o incluso RPCs, además hace falta un empaquetamiento de los datos. Se necesita un fichero que generara el compilador MIDL que contendrá la declaración de los interfaces descritos en el fichero IDL, y que habrá que incluir en la aplicación cliente, ya que en esta definición de interfaces se encontrarán todas las llamadas a procedimiento remoto, necesarias, así como el empaquetado de los parámetros.

2.6.2 Construcción real de un componente fuera del proceso

Si se ha seguido con atención el punto de construcción real de una aplicación, concretamente la parte en la que se explica las acciones que debía realizar una aplicación cliente para conectarse a un componente fuera del proceso, el presente punto se asimilará rápidamente.

Aunque en el servidor exista un archivo que contiene la definición y la implantación de las interfaces, será necesario la existencia de LPCs o RPCs y el empaquetamiento de parámetros para el paso de datos entre el servidor y aplicación cliente. Es por esto por lo que se deberá incluir el archivo que generó el compilador MIDL con la definición de las interfaces junto con las LPCs necesarias.

Por último, y para hacer las cosas más homogéneas, se suele incluir el archivo que generó el compilador MIDL. En él se encuentran los IIDs de las interfaces, por lo que no hará falta definirlos en el código del servidor ya que estarán incluidos en ese archivo. Lo único que habrá que definir será el CLSID de los componentes que implanta el servidor. Si esto se hace en un archivo independiente, éste podrá ser usado por la aplicación cliente para incluirlo en su código y así tener los CLSIDs de los componentes a utilizar.

3 JavaBeans

3.1 Introducción

Los JavaBeans definen un modelo de componentes de software para java; se puede crear una aplicación a partir de estos componentes de software ya creados con anterioridad; esto tiene como ventaja la reutilización de código. "Un Jarabean es un componente de software reutilizable que puede ser manipulado visualmente en una herramienta como por ejemplo (Visual J++ o JBuilder)".

Existen dos tipos diferentes de JavaBeans: los primeros son los bloques construidos que forman parte de una aplicación y que sirven como herramienta tipo para conectar los componentes, así como editar su apariencia; por ejemplo un botón de la AWT podría ser un componente JavaBean. El segundo caso se refiere a aplicaciones regulares que son insertadas dentro de documentos, por ejemplo una página Web.

Uno de los principales objetivos de la arquitectura JavaBeans, es proporcionar una plataforma neutral de componentes. Se proporcionan una implementación funcional completa en todas las plataformas, para que un JavaBean se anide dentro de otro JavaBean. Sin embargo, en un nivel alto, cuando la raíz del JavaBean se integra en una plataforma contenedor específica (como Word, Visual Basic o el navegador de Netscape) entonces la API de los JavaBeans deben integrarse en la plataforma local de la arquitectura de componentes [Monson].

Esto significa, por ejemplo, que en las plataformas Microsoft la API de JavaBeans debe ser puenteada a través de COM y ActiveX. Esto es posible convirtiendo el JavaBean como parte de un objeto Live o integrando un JavaBean con LiveConnect dentro del Navegador Netscape.

Entonces el JavaBean deber ser capaz de correr en un amplio rango de ambientes diferentes. Y en cada ambiente fuente debe ser capaz de disparar eventos e invocar los servicios de los métodos, como cualquier otro componente.

Sin embargo la necesidad de un puente para conectarse con otros modelos de componentes (principalmente OpenDoc, OLE/COM/ActiveX y LiveConnect) ha sido una de las restricciones en el diseño de la APIs de JavaBeans.

Cada JavaBean varía en la funcionalidad que soporta, pero todos ellos se unifican en las características que los distinguen, estas son [Valesky]:

- Soporte para "introspección" lo cual sirve para analizar como trabaja un JavaBean.
- Soporte para "personalizar" de manera que cuando un JavaBean se usa en alguna aplicación, el usuario puede personalizar la apariencia y comportamiento de un JavaBean.
- Soporte para "eventos" que se utiliza para implementar una comunicación simple que puede ser usada para conectar los Javabeans.
- Soporte para "propiedades" que sirve para la personalización y el uso programático.

- Soporte para “persistencia” que es útil cuando un JavaBean es personalizado en alguna aplicación y entonces se debe salvar el estado de la personalización para que más tarde pueda ser recargado.

Un JavaBean invisible no requiere heredarse desde una clase base o interfaz particular; pero un JavaBean visible debe heredarse del componente java.awt.Component para que de esta manera pueda agregarse a un contenedor visual.

Comparando entre JavaBeans y bibliotecas de clases podemos darnos cuenta que no todos los módulos de software deberían ser desarrollados como JavaBean. Los JavaBeans son apropiados para componentes de software que pueden ser manipulados y personalizados visualmente, para realizar algún efecto. Las bibliotecas de clases son apropiadas para proporcionar funcionalidad que sea útil para los programadores pero que no dan el beneficio de una manipulación a través de una herramienta visual.

Comparando ahora tiempo de diseño contra tiempo en ejecución, cada componente JavaBean tiene que ser capaz de correr en diferentes ambientes. Hay realmente una serie de diferentes posibilidades, pero hay dos puntos importantes.

El primero es que un JavaBean debe ser capaz de ejecutarse en una herramienta de diseño de aplicaciones o ambiente de diseño. Este ambiente de diseño es muy importante, ya que implica que el JavaBean debe proporcionar información de diseño para el constructor de la aplicación y permitir al usuario final personalizar la apariencia y comportamiento del JavaBean.

El segundo, es que todo JavaBean debe ser usado en tiempo de ejecución con la aplicación generada. En este ambiente hay mucho menos necesidades de diseño de la información o de personalización.

En cuestiones de seguridad los JavaBean están sujetos al modelo de seguridad de Java. Específicamente, cuando un JavaBean corre como parte de un applet no confiable, estará sujeto a las restricciones de seguridad del applet y no podrá leer o escribir archivos o conectarse a servidores en la red. Sin embargo, cuando un JavaBean corre como parte de una aplicación “stand-alone” o como parte de un applet confiable, entonces es tratado como una aplicación de Java normal y se le permite el acceso a archivos y la conexión con servidores de la red.

Los desarrolladores pueden diseñar sus JavaBeans para correrlos como parte de un applet no confiable. Las principales características de la API de JavaBeans son:

- **Introspección**. Los desarrolladores de JavaBean tienen acceso ilimitado en tiempo de diseño, pero un acceso más limitado en un ambiente a tiempo de ejecución. Un ejemplo es, el administrador de seguridad que permitirá el acceso de aplicaciones confiables aún a métodos y atributos privados, pero en applets no confiables solo permitirá el acceso a métodos y atributos públicos.
- **Persistencia**. Los JavaBeans deben poder ser serializados o deserializados, ya sea en tiempo de diseño o en tiempo de ejecución. Sin embargo, en un ambiente a tiempo de ejecución, el JavaBean debe esperar el flujo de serialización para ser creado y controlado por su aplicación padre y no deben

asumir que ellos pueden controlar los datos serializados que se leen desde o escriben a. Así un navegador obliga el uso de la serialización para leer el estado inicial de un applet no confiable.

- **Interfaz Grafica de Usuario**. En general, en un applet no confiable no está permitida la ejecución de algún tipo de combinación de interfaz gráfica con su aplicación padre.

Ninguna de las restricciones anteriores aplica a los JavaBean que están corriendo como parte de aplicaciones Java, donde no tienen restricción de acceso.

Los componentes JavaBean deben ser activados localmente, es decir que dichos componentes deben correr en el mismo espacio de direcciones como su contenedor.

Así por ejemplo, si el contenedor es una aplicación de Java, entonces el contenido del JavaBean está corriendo en la misma máquina virtual de java que el contenedor. Si el contenedor es una aplicación no Java, entonces el JavaBean correrá en una máquina virtual de Java directamente asociada a la aplicación. (Normalmente esta máquina virtual estará corriendo en el mismo espacio de direcciones de la aplicación).

Para un enlace de JavaBeans hacia un sitio remoto, debemos tomar en cuenta que la arquitectura de los JavaBeans está diseñada para trabajar de una manera adecuada en un ambiente de Internet, es decir en un ambiente distribuido. Una parte clave del diseño de sistemas distribuidos es su ingeniería. Una buena división entre procesamiento local y remoto. El procesamiento local puede beneficiar la rápida comunicación con una máquina simple, mientras que el acceso remoto puede tener grandes retardos y sufrir una variedad de fallas de comunicación. Los desarrolladores de sistemas distribuidos tienden a diseñar sus interfaces remotas muy cuidadosamente, para minimizar el número de interacciones remotas y para usar varios tipos de cache de datos y actualizaciones por lotes para reducir el tráfico remoto.

Por lo tanto, los JavaBeans no tienen un mecanismo para la comunicación remoto, lo único que ellos proporcionan son varios mecanismos alternativos que permiten a los desarrolladores la conexión desde y hacia servidores de red.

Los tres principales mecanismos para acceso a redes, los proporciona la plataforma de java estos son:

- **RMI**. La invocación de métodos remotos facilita el desarrollo de aplicaciones distribuidas en Java. Las interfaces con los sistemas distribuidos pueden ser diseñadas bajo el modelo cliente/servidor para ser implementadas en las interfaces creadas. RMI llama de manera transparente los métodos solicitados; es decir, RMI se encarga de localizar el servidor donde se encuentra el método que se desea ejecutar desde el cliente.
- **IDL**. El sistema IDL lo implementa el estándar industrial OMG CORBA; esta organización estandariza el modelo de objetos distribuidos. Todas las interfaces del sistema se definen en el lenguaje de definición de interfaces. Los stubs (cabos) pueden ser generados desde esas interfaces IDL, permitiendo a los clientes JavaBean hacer llamadas al servidor y viceversa. CORBA proporciona multilenguaje, es decir que es un ambiente distribuido multi-vendedor y el uso

de CORBA le permite a los JavaBeans poder comunicarse con servidores que no están implementados en Java.

- **JDBC**. La API para el manejo bases de datos, es a través del estándar JDBC, esto permite que los componentes JavaBean puedan acceder a las bases de datos a través de sentencias SQL. Estas bases de datos pueden estar en la misma máquina o en servidores de máquinas remotas.

Otra solución para el cómputo distribuido es migrar objetos en la red. Esto es útil cuando se crean JavaBeans en una estación de trabajo que pueden obtener información en diferentes servidores que se encuentran en la red.

Muchos JavaBeans pueden tener una representación gráfica. Sin embargo, también es posible implementar JavaBeans invisibles es decir que no tiene una interfaz gráfica de usuario. Estos JavaBeans son capaces de llamar a métodos, disparar eventos, salvar su estado persistente, entre otras acciones; estos JavaBeans pueden ser usados como recursos compartidos con aplicaciones gráficas o con componentes en aplicaciones servidor que no tienen una apariencia gráfica.

Los JavaBeans deben asumir que están corriendo en un ambiente multihilo, es decir que varios hilos pueden simultáneamente ejecutar tanto eventos como métodos y/o conjuntos de propiedades.

Es responsabilidad del desarrollador de JavaBeans asegurar el comportamiento apropiado con el uso de multihilos. Para el caso de un JavaBean simple, esto puede ser manejado simplemente con hacer todos los métodos "Sincronizados".

3.2 Arquitectura de Comunicación de los JavaBeans (INFOBUS)

La extensa adopción del lenguaje de programación Java en la comunidad del Internet, abre una oportunidad para que los diseñadores creen un nuevo tipo de usos interactivos. Las especificaciones del lenguaje y del ambiente proporcionan los mecanismos para la creación y la administración de los sistemas reutilizables conocidos como JavaBeans. Sin embargo, las especificaciones no sugieren los métodos por los cuales estos JavaBeans pueden intercambiar datos dinámicamente.

El Infobus está diseñado para componentes que trabajan juntas sobre la misma máquina virtual Java. El diseño no está relacionado directamente con componentes que trabajan en máquinas virtuales de Java separadas, o que operen en diferentes procesadores.

En general, todos los JavaBeans cargados desde una clase dada pueden "ver" otros JavaBeans en el mismo cargador y hacer llamadas a métodos directos pertenecientes a esos JavaBeans. Los JavaBeans usan "introspección" para "aprender" o "descubrir" información acerca de los JavaBeans a tiempo de ejecución. Esto consiste en detectar el nombre de los métodos que son descubiertos a través de la introspección.

Las interfaces del Infobus permiten al diseñador de aplicaciones crear flujos de datos entre JavaBeans cooperativos. En contraste con el modelo de evento/respuesta, la semántica de interacción depende de un alto entendimiento del JavaBean específico. Cuando ocurre un evento y el componente responde a ese evento, las interfaces del

Infobus tienen muy pocas alternativas y tienen un conjunto invariante de llamadas a métodos para todos los componentes. La semántica de flujo de datos está basada en la interpretación del contenido de datos que se envían entre las interfaces del Infobus y no en los nombres o parámetros originados por los eventos, ni tampoco en los nombres o parámetros de retorno.

3.2.1 Tipos de componentes

Los JavaBeans asociados a una aplicación con Infobus pueden ser clasificados en tres tipos: productores de datos, consumidores de datos y controladores de datos. Un componente individual puede actuar como productor o consumidor de datos. Entre componentes, los flujos de datos son objetos conocidos como conceptos de datos Los controladores de datos son componentes especializados para mediar la interacción entre productores y consumidores.

3.2.2 Principales requerimientos para el Infobus

La estructura de una aplicación con Infobus debe reunir dos importantes requerimientos para el Infobus:

- El Infobus debe soportar la creación de aplicaciones interactivas sin requerir del soporte de una aplicación constructora. Es decir, los diseñadores de aplicaciones debe ser capaces de ensamblar esas aplicaciones usando herramientas de edición de páginas Web. Por lo tanto esas aplicaciones deben correr en ambientes que interpreten HTML estándar por ejemplo los navegadores, sin necesidad de extensiones específicas o soporte más allá del lenguaje Java.
- El Infobus debe tener soporte semántico para permitir la comunicación de datos en un formato canónico para el uso por múltiples consumidores. Un formato canónico incluye la codificación de datos (números, cadenas etc.) y navegación de estructura de datos (renglones, columnas, tuplas, etc.).

3.2.3 Protocolo Infobus para intercambio de datos

Paso 1. Membresía—Establecimiento del Infobus

Cualquier componente de Java puede conectarse al Infobus. Esto se hace implementando un InfobusMember, obteniendo una instancia de Infobus y finalmente se une como “Member”.

Paso 2. Escuchando Eventos para el Infobus

Una vez que un objeto es miembro del Infobus, recibe notificaciones por medio del Bus, para esto hay que implementar una interfaz y registrarla con el InfoBus. Existen dos interfaces de escucha de eventos definidas para soportar dos tipos básicos de aplicaciones InfoBus. Un consumidor de datos recibe un anuncio acerca de los datos

que están disponibles agregando un consumidor oyente al Bus. De manera similar un productor de datos recibe una petición de datos agregando un productor oyente al Bus.

Paso 3. Lugar en donde los datos son intercambiados

En el modelo de Infobus, los productores de datos notifican de la disponibilidad de nuevos datos que están listos para ser leídos (por ejemplo leer una URL, completar un cálculo). Los consumidores solicitan datos del productor como por ejemplo: inicialización de una applet, evento de un botón, etc... La identificación se realiza por medio del nombre de los datos, es decir, el desarrollador de la aplicación define los nombres para los conceptos dato través de los cuales se permite el intercambio de datos.

Por lo tanto, todos los productores y consumidores de datos deben proporcionar algún mecanismo para los diseñadores de aplicaciones, para que especifiquen el nombre de los conceptos dato para que a través de ellos se puedan intercambiar los datos.

Paso 4. Recuperar del código los datos

Un conceptos dato puede ser recuperado como una cadena o un objeto Java. Los objetos Java son típicamente objetos enlazadores de tipos primitivos como el Double o instancias o de otro núcleo de clases como una Colección. El propósito es tener un entendimiento especializado de los formatos de datos en la parte del consumidor de datos.

3.3 Seguridad de los JavaBeans

3.3.1 Características del lenguaje Java

Lenguaje simple. Java es un lenguaje de aprendizaje muy rápido. Es fácil escribir applets de interés desde el inicio. Todos aquellos programadores que alguna vez han utilizado C++ se darán cuenta que el lenguaje de Java es más sencillo, porque se han eliminado características que complican el aprendizaje en C, como son los apuntadores. Ya que Java tiene mucho parecido con C y C++ y la mayoría de personas los conocen de manera elemental, con estos conocimientos previos les resultará más fácil aprender Java. Los programadores que tienen experiencia en C++ pueden cambiar muy rápido a Java y ser buenos programadores en muy poco tiempo.

Orientado a objetos. El diseño de Java fue conceptualizado como un lenguaje orientado a objetos desde el inicio. Los objetos son estructuras que encapsulan sus datos así como los métodos conocidos en C como funciones que manejan los datos. El futuro de Java seguirá con la programación orientada a objetos, pero principalmente basados en red.

Distribuido. Java ofrece un conjunto de clases que pueden ser usadas en aplicaciones donde intervengan redes, un ejemplo son los sockets que permiten establecer y aceptar conexiones con servidores o clientes remotos, de esta forma se facilita la creación de aplicaciones distribuidas.

Interpretado y compilado a la vez. Cuando Java es compilado, el código fuente se transforma en un código muy parecido a ensamblador llamado bytecodes.

Cuando se interpretan los bytecodes, esto permite la ejecución de manera directa en cualquier máquina que tenga una máquina virtual de Java o por ejemplo en Internet se necesita contar con el objeto Java Runtime, para la ejecución en tiempo real.

Robusto. Para crear software altamente confiable, Java proporciona varias comprobaciones a tiempo de compilación y a tiempo de ejecución. Al no manejar apuntadores esto libera de algunos errores del programador; ahora para liberar memoria utilizada por Java existe un mecanismo llamado recolector de basura ("Garbage Collector") y de esta manera evita al programador tener que liberar memoria de manera explicita, ya que este recolector lo hace por él.

Seguro. Observando que Java tiene un enfoque distribuido, porque tomando en cuenta que los applets se descargan desde cualquier punto de la Red, surge la necesidad de implantar seguridad. De otra manera cualquier persona que descargue un applet tendría acceso completo a su computadora, como por ejemplo leer o borrar sus archivos, bloquear sus dispositivos y así. De esta forma se implementaron barreras de seguridad en el lenguaje y en el sistema de ejecución a tiempo real.

Independiente de la arquitectura. Una característica muy importante de Java es que puede ser ejecutado en muchas plataformas y sistemas operativos como Unix, Linux, Windows NT/XP/2003 y Macintosh. Para hacer esto posible, el compilador de Java genera bytecodes, esto es un formato intermedio diferente a la arquitectura, diseñado para transportar el código de manera eficiente a muchas plataformas de hardware y software.

Portabilidad. La independencia respecto a la arquitectura representa una parte de su portabilidad. Java especifica tamaños de tipos de datos y el comportamiento de sus operadores aritméticos, de manera que los programas funcionen en todas las plataformas. Estas dos últimas características se traducen en la Máquina Virtual Java (JVM).

Multihilo. En la actualidad las aplicaciones que solo puede ejecutar una tarea a la vez no son bien vistas; ahora se trata de explotar más las capacidades del procesador y surge la necesidad de múltiples hilos de ejecución ("multithreading") a nivel de lenguaje; especialmente útiles en la creación de aplicaciones distribuidas de red. Un ejemplo podría ser que mientras un hilo pide los datos de un usuario, otro hilo estará abriendo la conexión hacia la base de datos para la comparación de los usuarios y el control de acceso.

Dinámico. Java tiene dos etapas, que se denominan tiempo de compilación y tiempo de ejecución; a esta última también se le conoce como de ejecución en tiempo real. Las clases de Java se enlazan a medida que se necesita.

Applets. En Java se pueden crear dos tipos de programas: aplicaciones independientes y applets. Las aplicaciones independientes son como cualquier otro programa escrito en algún lenguaje. Los applets son programas que van inmersos en las páginas HTML. A

través de ellos se pueden manejar tanto textos como gráficos y se cuenta con la capacidad de ejecutar acciones, como: animar imágenes, establecer conexiones de red, presentar menús y cuadros de diálogo y luego ejecutar otras acciones.

3.3.2 Funcionamiento de Java

El archivo del código fuente puede ser escrito mediante cualquier editor ASCII (por ejemplo el bloc de notas o, con el editor suministrado con el paquete del lenguaje Java.

Una vez creado el archivo .java, se compila y se genera un archivo intermedio cuyo contenido es el de los llamados bytecodes y tiene la extensión .class.

Una vez generado el archivo .class, entonces esta listo para ser interpretado en cualquier máquina virtual de Java. Tratándose de applets, este archivo se descarga desde la Red. La máquina virtual Java es la que se encarga de interpretar los bytecodes y generar las instrucciones para que se ejecuten en la arquitectura sobre la que está instalada.

El modelo de las cuatro capas. El modelo de seguridad de Java se conoce como modelo de caja de juegos ("*Sandbox model"*), este modelo consiste en hacer todo lo permitido por java dentro del lenguaje mismo pero fuera nada es permitido.

Este modelo esta construido por cuatro barreras de defensa [vmspec]:

- Características del lenguaje/compilador
- Verificador de código de bytes
- Cargador de clases
- Gestor de Seguridad

Es necesario tener en cuenta que cuando se habla de barreras de defensa se refiere a un conjunto de cuatro defensas que al penetrar a una de ellas, el sistema queda en manos del enemigo. No es correcto pensar que para que el sistema caiga en manos enemigas deba vencer a las cuatro barreras de defensas. Por lo tanto podemos darnos cuenta que tenemos que cuidar las cuatro defensas para no permitir intrusiones al sistema.

Comenzaremos con el análisis de la barrera de defensa llamada características del lenguaje.

Características del lenguaje/compilador. Java fue diseñado deseando las siguientes características:

- Evitar errores de memoria, es decir, evitar el acceso a áreas de memoria restringidas.
- No permitir el acceso al sistema operativo
- Evitar que se bloquee la máquina sobre la que corre

Para lograr estos objetivos, se implementaron las siguientes características:

Evitar la utilización de apuntadores. Con esto se evita la clonación de objetos, no permite la violación del encapsulado de los objetos, acceso a áreas protegidas de

memoria, esto se logra porque el programador no podrá hacer referencia de posiciones de memoria específicas que no fueron reservadas, lo anterior si se puede hacer en C y C++ y con ello se puede tirar la máquina anfitrión.

Gestión de memoria. No se puede gestionar la memoria de forma tan directa como en C, (no hay un recurso parecido a malloc). En cambio, se instancian objetos, el gestor de memoria no reserva memoria de manera directa, utiliza la instrucción new, la cual regresa una referencia, y de esta manera minimiza la interacción del programador con la memoria y con el sistema operativo.

Recolector de basura. Este es un gran avance del lenguaje, porque el programador no tiene que liberar la memoria de forma manual como se hace en C mediante la instrucción free. El recolector de basura de Java se encarga de solicitar que se libere la memoria usada por un objeto, una vez que éste ya no es accesible o desaparece. De esta manera, al ceder parte de la gestión de memoria a Java, se evitan errores de memoria y los apuntadores que quedan sin referencia también son recolectados.

Arreglos con comprobación de límites. En Java los arreglos son objetos, lo cual les confiere ciertas funciones muy útiles, como la comprobación de límites. Para cada subíndice, Java comprueba si se encuentra en el rango definido, según el número de elementos del arreglo, previniendo así que se haga referencia a elementos que estén fuera del límite.

Conversiones seguras. Sólo se permiten conversiones entre ciertas primitivas del lenguaje (ints, longs) y entre objetos de la misma rama del árbol de herencia.

Control de métodos y variables de clase. Las variables y los métodos declarados privados sólo son accesibles por la clase o subclases herederas de ella y los declarados como protegidos, sólo por la clase.

Métodos y clases final. Las clases y los métodos declarados como final no pueden ser sobrescritos o modificados. Una clase declarada final no puede ser heredada.

Verificador de ByteCodes. Sólo permite ejecutar código de bytes de programas Java válidos, tratando de detectar intentos de:

- Crear apuntadores.
- Ejecutar instrucciones en código nativo.
- Llamar a métodos con parámetros no válidos.
- Usar variables antes de inicializarlas.

El verificador efectúa cuatro pasadas sobre cada archivo de clase:

- En la primera, se valida el formato del archivo
- En la segunda, se comprueba que no se instancien subclases de clases final
- En la tercera, se verifica el código de bytes: la pila, registros, argumentos de métodos, opcodes
- En la cuarta, se finaliza el proceso de verificación, realizándose las últimas pruebas

Si el verificador aprueba un archivo .class, se supone que ha cumplido con los siguientes requisitos:

- Acceso a registros y memoria válidos
- No hay desbordamientos de pila
- Consistencia de tipo en parámetros y valores devueltos
- No hay conversiones de tipo ilegales

Cargador de Clases. En el momento de la ejecución de los applets, en nuestra computadora se consideran tres formas de ejecución con diferentes niveles de seguridad:

- La máquina local, este es el más seguro.
- La red local resguardada por el "firewall", es considerado como seguro porque se tiene una barrera de seguridad.
- Internet, este medio se considera como inseguro porque al descargar recursos en nuestra computadora no sabemos que operaciones secretas van a realizar; por ejemplo sacar información de nuestra computadora entre otras.

En este contexto, no es válido permitir a una clase con una seguridad inferior, sustituir a otra de una seguridad superior; ello con el fin de evitar que una applet cargue una de sus clases para reemplazar a una clase crítica del sistema, permitiendo con esto degradar las restricciones de seguridad de esa clase.

Este tipo de ataque se cubre asignando un espacio de nombres diferente para clases locales y para clases cargadas desde la red. Siempre se accede antes a las clases del sistema, en lugar de clases del mismo nombre cargadas desde una applet.

3.3.3 Administrador de Seguridad

La gestión de seguridad la realiza la clase abstracta SecurityManager, que limita lo que las applets pueden o no hacer. Para prevenir que sea modificada por alguna applet maliciosa, esta clase no puede ser heredada por las applets.

Entre sus funciones de vigilancia, se encuentran el asegurar que las applets no acceden al sistema de archivos, no abran conexiones a través de Internet, no acceden al sistema, etc. Para ello, va a introducir una serie de restricciones de seguridad o prohibiciones.

3.3.3.1 Restricciones de seguridad.

Es necesario tomar en cuenta que cada navegador ofrece su propia política de seguridad. La mayoría de los navegadores presentarán las siguientes restricciones en el momento de ejecutar los applets:

- No se puede trabajar con el sistema de archivos: leer, escribir, borrar, renombrar, listar, conseguir información, ejecutar programas, etc.

- No se pueden establecer conexiones de red a máquinas distintas que la que envió el applet
- No se permite acceso al sistema
- No se permite manipulación de hilos (“threads”)
- No se pueden cargar métodos nativos
- No se pueden evitar mensajes de alerta en las ventanas creadas por el applet
- No se pueden crear subclases de SecurityManager en una applet
- No se permiten puertas traseras (“*backdoor”*).

3.3.3.2 Restricción de acceso al sistema de archivos.

Es claro comprender por qué se le niega el acceso a nuestro sistema de archivos a cualquier applet desconocida, descargada desde la red. Si no existiera esta restricción, las applets plantearían los siguientes problemas:

- Acceso a información de seguridad: como por ejemplo el archivo /etc/passwd de Unix.
- Acceso a información privada: correo, documentos, etc., almacenados en nuestro disco duro.
- Si las applets pudieran escribir ficheros, podrían introducir virus (especialmente caballos de Troya, como aplicaciones aparentemente de utilidad, pero que en realidad abrirían la puerta a problemas).
- Al escribir un archivo se puede llegar a llenar el disco, causando error al querer grabar nuestros datos.
- Se podrían modificar los archivos (por ejemplo. cifrarlos o destruirlos).
- Se pueden crear archivos de engaño.
- Conocer la estructura de directorios puede revelar información e identificar debilidades acerca del sistema (especialmente en Unix):
    - Identificar nombres de programas CGI ocultos.
    - Identificar debilidades de seguridad.
    - Revelar archivos .rhost.

3.3.3.3 Restricción de conexiones de red.

Si las applets fueran capaces de establecer conexiones sin ningún tipo de control, se podrían producir los siguientes tipos de ataque:

- Desde dentro de un “firewall”: el applet hostil podría descargarse desde una inocente página Web, pasando los controles del firewall, y una vez dentro, hacerse pasar por la máquina que lo ha descargado para establecer conexiones o recibirlas.
- Suplantación de usuario, explotando la confianza de otras máquinas: una vez que el usuario incauto ha descargado inadvertidamente la applet y se encuentra en ejecución en su sistema, la applet podría comunicarse con otras máquinas usando la dirección IP de la máquina del usuario y enviar por ejemplo correos ofensivos.

- Escucha de puertos que suplanten a un servicio (p.e. servidor Web, escuchando en el puerto 80 y respondiendo a todas las peticiones de páginas con sus propias páginas. Algo terrible para el sitio Web suplantado).

Para evitarlo, el gestor de seguridad se cuida de comprobar todos los intentos de conexión. En concreto, la clase SecurityManager define los siguientes métodos mostrados en la tabla 3.1.

Método	*Descripción*
CheckAccept(String, int)	Se llama antes de aceptar una conexión con un socket de la máquina especificada en el puerto indicado.
CheckConnect(String, int)	Se llama antes de abrir una conexión con un socket a la máquina especificada en el puerto indicado.
CheckConnect(String, int, Object)	Se llama antes de abrir una conexión con un socket a la máquina especificada en el puerto indicado, para el contexto de seguridad actual.
checkListen(int)	Se llama para determinar si el proceso en curso tiene permiso para quedarse a la escucha de conexiones en el puerto especificado.
checkSetFactory()	Determina si el proceso en curso tiene permiso para establecer la fabricación de manejadores de socket o streams de URL.

Tabla 3.1. Métodos de la clase SecurityManager.

3.3.3.4 Restricción de acceso al sistema.

A las applets se les niega la capacidad de ejecutar programas en la máquina local y de acceder a propiedades del sistema, ya que estas capacidades podrían usarse para robo o destrucción de información o denegación de servicio:

- Terminar la sesión del navegador (simplemente llamando a System.exit()).
- Ejecutar comandos que manipulen los ficheros u otros servicios.
- Conocer información privada del sistema o que puede revelar debilidades.

3.3.3.5 Restricción de de manipulación de hilos (“threads”).

Las applets podrían destruir trabajo cerrando o inhabilitando otros componentes de las aplicaciones en que corren ataques de denegación de servicio, aumentando su propia prioridad.

3.3.3.6 Restricción de acceso a métodos nativos.

Se denomina como métodos nativos a código escrito en otro lenguaje distinto de Java, como en C o en ensamblador. Las bibliotecas de métodos nativos, por un lado, resultan de crucial importancia para extender la funcionalidad base de la Máquina Virtual Java; por otro lado, constituyen la última puerta trasera de entrada, ya que pueden sortear los

mecanismos de seguridad de Java, aunque no tienen por qué. Por este motivo, cargar una biblioteca de métodos nativos podría comprometer la seguridad de todo el sistema, ya que al no estar escritos en Java, no necesitan respetar las reglas de protección de Java y ni siquiera se les exige el usar las clases existentes. Así, pueden acceder a los recursos locales del sistema, bien proporcionando acceso a nuevos recursos no contemplados en Java y sin protegerlos adecuadamente o bien burlando las comprobaciones de seguridad de Java.

3.3.3.7 Restricción de creación de ventanas.

Es evidente la razón por la cual a las ventanas creadas desde una applet se le añaden una serie de advertencias acerca de su inseguridad, como por ejemplo en la parte inferior de un applet aparece el siguiente mensaje applet de java, si no existiera ninguna señal de alerta, podrían suplantar ventanas de aplicaciones locales de confianza, como la conocida ventana de Windows 95 ó NT que piden el nombre de usuario y la contraseña.

Existen muchos tipos de ataques a la seguridad que podrían organizarse contra un sistema de computadoras. Quizá un usuario que no tiene mucha experiencia podría pasar por alto que tales amenazas existen, lo cierto es que los ataques suceden, a veces con consecuencias muy graves. A continuación los principales ataques y cómo se defiende Java de ellos.

Entre los ataques más comunes citaremos:

- Robo de recursos
- Enmascaramiento
- Engaño
- Destrucción de información
- Denegación de servicio
- Robo de información

Otra clasificación de ataques que se pueden encontrar en la literatura más relacionada con Java es la siguiente:

- Modificación o alteración de un sistema o sus recursos.
- Denegación del uso legítimo de los recursos de la máquina.
- Ataques a la intimidad del individuo.
- Por molestar.

3.3.3.8 Robo de información.

En principio, todas las computadoras contienen alguna información de interés. Es cierto que no siempre tendrá el mismo valor, pero siempre puede existir alguien interesado en conseguirla. Por consiguiente, uno de los ataques más comunes está dirigido a extraer información de un sistema. Con el fin de que ningún applet de Java pudiera hacerlo, se decidió que no tendrían acceso al sistema de archivos, lo que significa que no se puede leer, escribir, borrar, renombrar o listar archivos. Además se les negó el acceso al sistema, para que no pudieran ejecutar comandos que manipulen ficheros (pensemos en el famoso rm -rf / de Unix) ni acceder a propiedades del sistema que contengan

información confidencial (nombres de usuario, contraseñas, versiones de programas, etc.) o que pudieran revelar vulnerabilidades.

En concreto, el gestor de seguridad define los métodos públicos de la clase SecurityManager, que se muestran en la tabla 3.2, para evitar estos ataques:

Método	***Descripción***
checkDelete(String)	Comprueba si se puede borrar el archivo de nombre String.
checkRead (FileDescriptor)	Comprueba si se puede leer el archivo indicado por FileDescriptor.
checkRead(String)	Comprueba si se puede leer el archivo de nombre String.
checkRead(String, Object)	Comprueba si se puede leer el archivo en el contexto de seguridad actual.
checkWrite (FileDescriptor)	Comprueba si se puede escribir el archivo indicado por FileDescriptor.
checkWrite(String)	Comprueba si se puede escribir el archivo de nombre String.
checkFileDialog()	Las applets no pueden abrir cuadros de diálogo de archivos.
checkExec(String)	Determina si se puede crear un subproceso para el comando especificado.
checkExit(int)	Determina si el proceso en curso puede detener la Máquina Virtual Java (JVM) con el código de salida especificado.
CheckPropertiesAccess()	Determina si se puede acceder a las propiedades del sistema.
CheckPropertyAccess (String)	Determina si el proceso en curso puede acceder a la variable de entorno especificada.

Tabla 3.2. Métodos de la clase SecurityManager.

Pero existen métodos indirectos que permiten explorar la información del disco duro.

3.3.3.9 Destrucción de información.

Todos poseemos datos y archivos en nuestra computadora que no quisiéramos perder. Por estas razones es muy importante que ninguna applet pueda modificar y mucho menos destruir los datos de la máquina cliente en la que se ejecuta. Los diseñadores de Java introdujeron una serie de controles de seguridad que impiden el acceso al disco. Estas funciones se muestran en la tabla 3.3.

Método	***Descripción***
checkDelete(String)	Comprueba si se puede borrar el archivo de nombre String.
checkRead (FileDescriptor)	Comprueba si se puede leer el archivo indicado por FileDescriptor.
checkRead(String)	Comprueba si se puede leer el archivo de nombre String.
checkRead(String, Object)	Comprueba si se puede leer el archivo en el contexto de seguridad actual.
checkWrite (FileDescriptor)	Comprueba si se puede escribir el archivo indicado por FileDescriptor.
checkWrite(String)	Comprueba si se puede escribir el archivo de nombre String.
checkFileDialog()	Los applets no pueden abrir cuadros de diálogo de archivos.

Tabla 3.3. Controles de seguridad que impiden el acceso al disco.

3.3.3.10 Robo de recursos.

Las computadoras, además de almacenar datos, poseen valiosos recursos como espacio en disco y en memoria, ciclos de CPU, capacidad de cálculo, etc. Un applet descargado silenciosamente desde una página Web podría estar trabajando en bajo nivel (“background”), sin hacer ruido, realizando costosos cálculos y enviando los resultados al servidor desde el que bajó si que el usuario se diera cuenta.

Es éste el aspecto más difícil de controlar en Java, ya que los applets pueden hacer uso ilimitado de ciclos de CPU, en algunos casos esto vuelve lento el funcionamiento de la computadora.

3.3.3.11 Denegación de servicio.

En general, estos ataques se basan en utilizar la mayor cantidad posible de recursos del sistema en el cual se están ejecutando, de manera que nadie más pueda usarlos, perjudicando así seriamente el rendimiento del sistema; especialmente si debe dar servicio a muchos usuarios.

Ejemplos típicos de este ataque son:

- Abrir en forma de cascada muchas ventanas para saturar el accionar del teclado o de los botones del ratón; y que de esta manera ya no se pueda utilizar la computadora, porque tendrían que cerrarse todas las ventanas para poder trabajar.
- Consumir toda la memoria de la computadora y cuando se termine enviar un mensaje de error en donde se indica falta de memoria.
- Bloquear algún periférico, para que nadie lo pueda utilizar, de esta forma se obliga a que sea desconectado o que termine algún proceso que corre en la computadora.
- Consumir rebanadas de tiempo asignadas por el CPU, de tal manera que los demás procesos dispongan del mínimo tiempo de ejecución y esto da como resultado que el rendimiento del sistema decaiga rápidamente.
- Bloquear algún periférico, para que nadie lo pueda utilizar, de esta forma se obliga a que sea desconectado o que finalice con algún proceso que corre en la computadora.

Desafortunadamente, Java no tiene una política que permita evitar estos ataques, ya que resulta muy complicado determinar el límite de la cantidad de recursos que un applet tiene derecho a reservar para sí mismo.

3.3.3.12 Engaño.

Un ataque muy común consiste en engañar a una persona para que nos facilite información confidencial. Un ejemplo típico es el que se conoce como de ingeniería social: te haces pasar por el administrador de una red o como personal de servicio y

pides a los usuarios incautos que te envíen su nombre de usuario y contraseña, por ejemplo para hacer una actualización de la versión de un programa X.

Una forma más sofisticada de engaño consiste en falsificar ventanas del sistema, para obligar al usuario a que ejecute una acción que de otro modo no realizaría, como reiniciar la computadora (perdiendo los datos) o escribir su nombre de usuario y contraseña.

Existen formas de hacerlo en Java, los diseñadores de Java han luchado contra esto, pero aunque incluyeron la clase SecurityManager no lo han podido evitar.

Método	***Descripción***
boolean checkTopLevelWindow (Object)	Determina si se puede mostrar una ventana de aplicación sin añadir restricciones, normalmente un aviso de que la ventana pertenece a un applet.

3.3.3.13 Futuro de la seguridad en Java.

Actualmente se trabaja en el API de Seguridad de Java, que tiene como meta proporcionar servicios de validación y cifrado para las aplicaciones Java. Por supuesto, dada la extrema dependencia de la seguridad a lo que se decida en este API, ha surgido una gran polémica en torno a qué medidas de seguridad adicionales deberían incorporarse.

Un avance que han tratado de lograr es que los applets puedan ser firmados para determinar que no son malignos, esta firma la debe dar una autoridad de certificación. De esta forma, cuando el applet es descargado en la computadora cliente, presenta una firma digital, para que el usuario pueda decidir si lo descarga o no. Los applets seguros tendrán el privilegio de acceder de forma controlada a los recursos, mientras que los no firmados tendrán muchas restricciones para acceder a los recursos del sistema.

3.3.3.14 Evolución del modelo de seguridad de Java.

Muchos desarrolladores de software se encuentran ante la restricción de escribir aplicaciones que sean realmente útiles, porque se les niega la capacidad de leer y/o escribir en disco o de realizar conexiones a otras computadoras; para que realmente los applets sean tomados en serio y no como simples "banners", los usuarios de aplicaciones Java necesitan asegurar el acceso a los recursos, pero para poder hacer esto es necesario un mecanismo que permita tener la seguridad de como y de donde fue descargado el applet, para de esta forma decidir los privilegios de acceso al sistema.

La primera versión de seguridad en Java, es proporcionad por la API para:

- **Gestión de claves:** es un conjunto de abstracciones para gestionar claves y certificados de entidades tales como usuarios individuales o grupos. Permite a las aplicaciones diseñar sus propios sistemas de gestión de claves e interoperar con otros sistemas a alto nivel.

- **Firmas digitales**: algoritmos de firma digital, como DSA. Su funcionalidad incluye la generación de pares llave pública/privada, así como firma y verificación de datos digitales diversos.
- **Listas de control de acceso**: son un conjunto de abstracciones para gestionar cuentas de usuario y sus permisos de acceso.
- **Resúmenes de mensajes criptográficamente seguros**: tales como MD5 y SHA-1. Estos algoritmos, también llamados algoritmos de hash unidireccional, son útiles para producir "resúmenes digitales" de los datos, usados frecuentemente en firmas digitales y otras aplicaciones que requieren identificadores únicos e infalsificables relacionados con los datos digitales.

De forma paralela se creó una Extensión Criptográfica de Java (JCE) con APIs que incluían algoritmos para cifrado y descifrado de datos.

3.3.3.15 Nueva arquitectura de seguridad de Java.

La nueva arquitectura de seguridad del JDK 1.2 se introdujo con el fin de superar limitaciones del modelo anterior:

- **Control de acceso de grano fino**: libera al programador de la carga de heredar y adaptar las clases SecurityManager y ClassLoader.
- **Extensión de las comprobaciones de seguridad a todos los programas en Java**, **incluyendo tanto aplicaciones como applets**: el código local ya no es de confianza, sino que también está sujeto a los controles de seguridad, aunque estas políticas pueden flexibilizarse.
- **Política de seguridad fácilmente configurable**: permite a los desarrolladores y usuarios configurar políticas de seguridad sin necesidad de programar una línea.
- **Estructura de control de acceso fácilmente extensible**: acaba con la necesidad de añadir nuevos métodos a la clase SecurityManager para crear nuevos permisos de acceso, ya que permite hacerlo automáticamente.

Lo fuerte de esta nueva arquitectura es el concepto de dominio de protección, mecanismo para agrupar y aislar unidades de protección; a los cuales se les asigna una serie de permisos de acceso a recursos. De esta manera, cada clase pertenece a un único dominio, que tendrá sus permisos asignados. Antes de acceder a un objeto, se comprueba la política para ver qué permisos tiene y si puede acceder al servicio requerido.

Otra característica es el soporte para SSL v3.0 (Secure Socket Layer) y una nueva extensión criptográfica para Java. La verificación de firmas digitales soporta completamente el procesamiento de los certificados X.509v3.

3.4 Vulnerabilidad de los Componentes JavaBeans

Los JavaBeans no implementan seguridad sino que se basan en la seguridad que le brinda la máquina virtual de Java, por lo tanto cualquier vulnerabilidad en Java se convertirá en un hueco de seguridad para los JavaBeans. Algunas de las

vulnerabilidades se deben a que las comprobaciones del cargador de clases no son suficientes respecto al acceso a paquetes [iblnews].

De hecho, debido a un error lógico en la implementación del método loadClass de la clase sun.applet.AppletClass, es posible cargar cualquier clase en la máquina virtual sin tener que realizar una llamada al método checkPagkageAccess del administrador de seguridad de Java (Security Manager).

La técnica consiste en sustituir los '.' por '/' a la hora de realizar el nombrado dentro del árbol de un paquete. En condiciones normales, por política estándar de seguridad, un acceso al paquete 'sun.' provocaría una excepción de seguridad. Sin embargo, la comprobación se hace verificando esos '.', así que debido a que de forma interna se utiliza también el carácter '/' para acceder a diferentes partes del árbol, se podría hacer una llamada a 'sun/paquete_arbitrario/clase_arbitraria' sin que se active el anteriormente nombrado mecanismo de seguridad.

Esta vulnerabilidad permitiría la creación de applets maliciosos que pueden saltarse completamente las restricciones del “sandbox”. Se han realizado experimentos con códigos de “prueba de concepto” que han tenido éxito explotándola en navegadores como Netscape 6 y 7 (además de Mozilla) que utilizan versiones vulnerables de dicha máquina virtual.

Las versiones vulnerables (en el SDK y JRE) son 1.4.1_03 y anteriores, 1.3.1_08 y anteriores, y 1.2.2_015 y anteriores.

Ya que la vulnerabilidad afecta a un componente tan importante, es conveniente reseñar que este problema puede afectar potencialmente a todos los usuarios que utilicen este componente en sus navegadores. Además de los anteriormente nombrados, otros como Opera e Internet Explorer podrían estar afectados, aunque no se ha confirmado.

Otras vulnerabilidades se presentan con el manejo de las bases de datos a continuación se mencionan las siguientes [virusprot]:

- La primera de las vulnerabilidades permite a un atacante conseguir el control total del sistema de otro usuario, debido a una vulnerabilidad de la Máquina Virtual que se ha detectado en Java Database Connectivity (JDBC), módulo cuya función es dar soporte a bases de datos.

- La segunda de las vulnerabilidades, también relacionada con JDBC, posibilita a un agresor causar una falla puntual en la versión de Internet Explorer instalada en el ordenador de la víctima.

- La tercera de las vulnerabilidades permite que un atacante logre el control del sistema de otro usuario aprovechando un problema existente en una función encargada de dar soporte al formato XML en las aplicaciones Java.

- Afortunadamente en la última versión del JDK ya fueron corregidas, pero aún seguirá la búsqueda de otros huecos de seguridad.

4 CORBA

4.1 Introducción a la arquitectura CORBA

Otra de las arquitecturas de software de especial importancia es CORBA la cual está orientada a objetos remotos. La especificación de CORBA ("Common Object Request Broker Architecture") define un modelo de objetos donde los clientes envían mensajes a objetos remotos solicitándoles la ejecución de alguno de sus métodos [OMG-2.2]. Objetos como los servidores están encapsulados, y por lo tanto, la representación de sus datos y la implantación de sus métodos queda oculta para los clientes. Cada objeto remoto tiene una interfaz que define los métodos públicos que pueden ser invocados por los clientes, especificando los tipos de datos que se usan como parámetros en la llamada, así como el tipo de datos del o de los resultados.

Cada objeto remoto tiene un identificador independiente de la localización, que usan los clientes cuando hacen sus peticiones; por lo tanto una invocación a un método remoto debe especificar el identificador del objeto, el método, y algunos parámetros. La respuesta incluye los resultados y posibles excepciones.

Las interfaces de CORBA no proveen métodos para crear o destruir objetos, pero los objetos pueden ser creados o destruidos como parte del efecto de la invocación de algún método. Los servidores generalmente contienen una colección de objetos, de los cuales uno o más pueden ser remotos. Cuando un servidor se encuentra en ejecución, puede crear nuevos objetos remotos y regresar sus identificadores a los clientes. Los objetos servidores pueden convertirse en clientes de otros objetos remotos.

4.2 Estructura de un agente de invocación de objetos

La arquitectura está basada en un agente de invocación de objetos (ORB: "Object Request Broker") que permite a los clientes invocar métodos de objetos remotos que pudiesen estar implementados en una amplia diversidad de lenguajes y localizados físicamente en alguno o algunos sitios conectados a la red. La figura 4.1(a) muestra una invocación enviada por un cliente a una implantación de un objeto, mediante la ayuda del agente de invocación de objetos. El cliente es la entidad que desea realizar una operación sobre el objeto y la implantación del objeto es el código y datos que implementan al objeto.

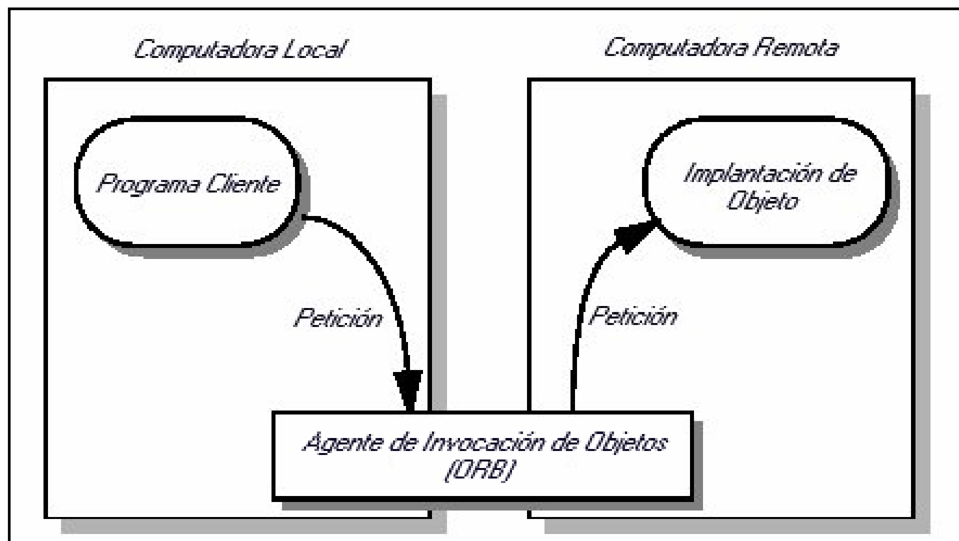
El ORB es responsable de todos los mecanismos necesarios para localizar al objeto que implementa el método invocado [Siegel] [Orfali], lo prepara para recibir la invocación, y le comunica los parámetros necesarios para que se lleve a cabo. La interfaz que el cliente ve es totalmente independiente de la localización del objeto, del lenguaje en que está implementado o de cualquier otro aspecto que no quede especificado en la interfaz del objeto. La figura 4.1 (b) muestra la estructura de un agente de invocación de objetos.

Para realizar una invocación, el cliente tiene esencialmente dos posibilidades:

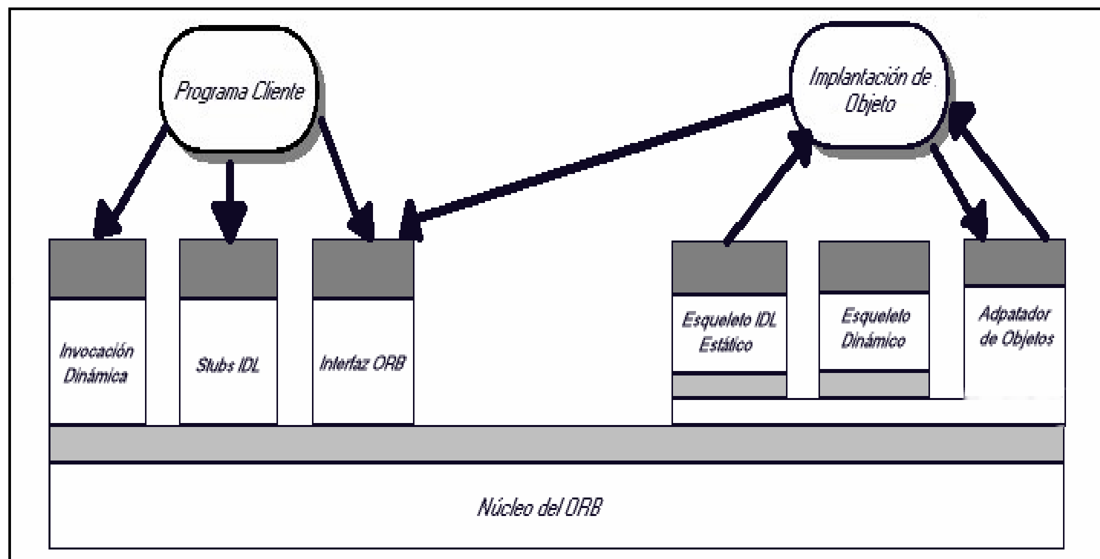
- Puede usar una interfaz de acceso dinámica (la misma interfaz, independiente de la interfaz del objeto destino)

- O mediante un cabo ("stub") generado por el lenguaje de definición de interfaces conocido como IDL (un cabo específico, dependiente de la interfaz del objeto destino).

Además los clientes pueden interactuar directamente con el ORB para realizar algunas funciones específicas. La implantación del objeto recibe peticiones, ya sea a través del esqueleto generado por el lenguaje de definición de interfaces o mediante un esqueleto dinámico. Es importante notar que la implantación del objeto tiene la posibilidad de realizar llamadas al adaptador de objetos y al ORB mientras está procesando una petición o en cualquier otro momento.



(a)



(b)

Figura 4.1. (a). Invocación enviada por un cliente a una implantación de un objeto. (b). Estructura de un agente de invocación de objetos (ORB).

Los clientes CORBA realizan sus peticiones mediante una referencia o identificador de objeto y, conociendo el tipo de objeto y el servicio que se desea recibir de él [Orfali-G] [Orfali]. El cliente inicia la petición mediante un llamado a las rutinas de cabo que son específicas del objeto o construyendo la petición de manera dinámica. Las interfaces cabo y dinámica satisfacen la misma semántica de invocación, es decir, proporcionan la misma información, de tal manera que el receptor final del mensaje no pueda decir con cual de las dos maneras fue invocado. Es el ORB quien se encarga de localizar el código que implementa al objeto, transmitirle los parámetros y transferirle el control ya sea a través del esqueleto IDL o del esqueleto dinámico.

Los esqueletos son específicos de la interfaz y del adaptador de objetos. Para procesar la petición, la implantación del objeto puede obtener servicios del ORB mediante el adaptador de objetos, y cuando se completa el servicio, el control y los valores de salida se regresan al cliente.

La implantación del objeto puede elegir qué adaptador de objetos utilizar, esta decisión está basada en el tipo de servicios que requiere la implantación del objeto [OMG-2.2]. La figura 4.2 muestra como la información referente a la interfaz y a la implantación se pone a disposición de los clientes e implantaciones de los objetos. La interfaz se define en el IDL y/o en el repositorio de interfaces; esta definición se utiliza para generar los cabos del cliente y los esqueletos de las implantaciones de los objetos. La información referente a la implantación del objeto se proporciona al momento de instalarla o darla de alta en el sistema, y se guarda en el repositorio de implantaciones para su uso durante la entrega de una petición.

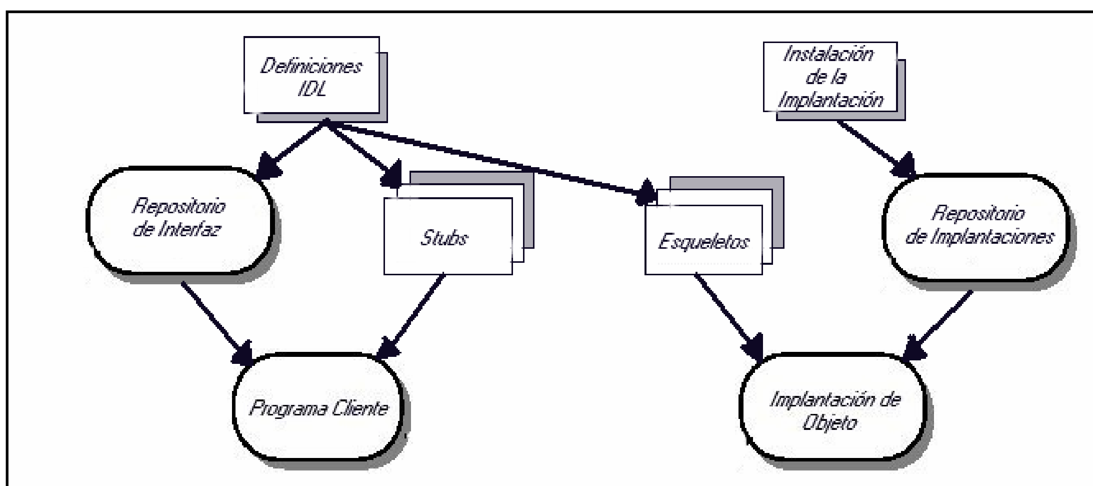


Figura 4.2. Repositorio de interfaz y de implantaciones.

4.2.1 Agente de invocación de objetos

En la arquitectura CORBA, el ORB no necesariamente debe ser implantado como un componente único, más bien está definido por sus interfaces. Por lo tanto, cualquier implantación de un ORB que proporcione las interfaces adecuadas es aceptable. La interfaz está organizada en tres categorías:

1. Operaciones comunes a todas las implantaciones de ORBs.
2. Operaciones específicas a algún tipo particular de ORB.
3. Operaciones específicas a algún tipo particular de implantación de objeto.

En la arquitectura se permite que los ORBs tengan diferentes elecciones para su implantación y junto con los compiladores IDL, repositorios y adaptadores de objetos, proporcionan diversos servicios a los clientes y a las implantaciones de los objetos. Es posible que existan múltiples implantaciones de ORBs que tengan diferentes representaciones para sus referencias de los objetos y diferentes medios para realizar las invocaciones de objetos [Vinoski]. También es posible que un cliente acceda a dos referencias de objetos manejadas por diferentes implantaciones de ORB; cuando esto sucede, es responsabilidad de los ORBs y no del cliente, distinguir a que referencia de objeto corresponde su invocación.

El núcleo del ORB es la parte que proporciona la representación básica de los objetos y la comunicación de peticiones [Andreas]. CORBA está diseñado para soportar diferentes mecanismos de objetos, y lo logra estructurando al ORB en una arquitectura de capas con otros componentes montados por encima del núcleo del ORB, proporcionando interfaces que enmascaran las diferencias entre los núcleos.

4.2.2 Clientes

Un cliente de un objeto es la entidad de software que tiene acceso a una referencia del objeto y, mediante ella realiza invocaciones a él. El cliente únicamente conoce la estructura lógica del objeto, de acuerdo con su interfaz y accede al comportamiento del objeto a través de invocaciones a sus métodos [Orfali-CS]. A pesar de que generalmente se considera al cliente como un programa o proceso que realiza invocaciones a un objeto, es importante notar que la implantación de un objeto también puede ser a su vez, cliente de algún otro objeto.

Los clientes perciben a los objetos y las interfaces del ORB mediante la perspectiva que les proporciona el mapeo a su lenguaje de programación, y por lo tanto, los programadores perciben al ORB al nivel de su lenguaje de programación. Los clientes son altamente portables y deben ser capaces de funcionar sin ningún cambio en su código fuente en cualquier ORB que soporte el mapeo al lenguaje en el que fueron escritos. Los clientes no tienen ningún conocimiento sobre los detalles de la implantación del objeto, sobre de qué adaptador de objetos están utilizando, o acerca de qué ORB utilizan para acceder a él.

Los clientes CORBA realizan sus peticiones mediante una referencia, y conociendo el tipo de objeto y el servicio que se desea recibir de él, inician la petición mediante un llamado a las rutinas o métodos del cabo (dependientes del tipo de lenguaje del cliente) que son específicos del objeto; o construyendo la petición de manera dinámica, como se muestra en la figura 4.3.

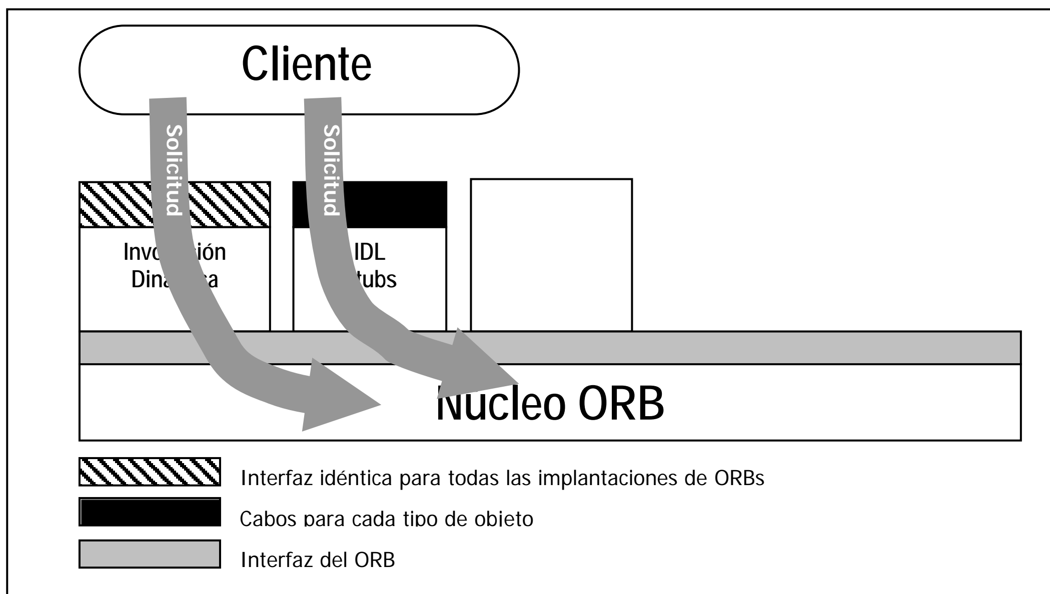


Figura 4.3. Cliente CORBA.

4.2.3 Implantaciones de objetos

La implantación de un objeto provee la semántica del objeto, generalmente mediante la definición de los datos de instancia y del código que implementa los métodos. Es común que la implantación de un objeto utilice otros objetos o algún componente de software adicional para implementar el comportamiento del objeto; y en algunos casos, la función primaria del objeto será la de inducir algún comportamiento en otra instancia que no necesariamente es un objeto.

La arquitectura soporta una amplia variedad de formas de implantar objetos, incluyendo servidores, bibliotecas, aplicaciones encapsuladas, bases de datos orientadas a objetos, entre otras. Es importante resaltar que no es imperativo implementar los objetos remotos en un lenguaje orientado a objetos, y que es posible hacerlo en lenguajes como C.

4.2.4 Referencias a objetos

La referencia de un objeto es la información necesaria para especificar al objeto dentro del ámbito de un ORB. Tanto clientes como implantaciones tienen una noción parcial en las referencias de los objetos, de acuerdo con el lenguaje que utilicen. Esta noción parcial se debe al hecho de que el mapeo de la referencia, dentro del lenguaje de programación, no permite observar dentro del ORB, todas las características que ésta posee [Geraghty], sino que muestra una visión parcial de ellas, en la cual algunos detalles de la implantación no pueden ser percibidos. Debido a esta percepción parcial de las referencias, los clientes están aislados de la representación real de la referencia contenida en el ORB.

Dos ORBs pueden diferir en el tipo de representación que utilicen para sus referencias a objetos, es decir, no es necesario que todas las implantaciones de ORB utilicen la misma representación interna para las referencias.

El ámbito de las referencias a los objetos abarca únicamente el tiempo de vida del cliente que recibió la referencia. Para un lenguaje de programación determinado, todos los ORBs deben proporcionar el mismo mapeo de una referencia de objeto; esto permite a un programa escrito en un lenguaje determinado, acceder a las referencias de los objetos, independientemente del ORB que se utilice. Para conveniencia del programador, el mapeo al lenguaje proporciona formas adicionales para acceder a las referencias de los objetos.

4.2.5 Cabos del cliente

La función de las rutinas del cabo es realizar el proceso de aplanado[6] de los argumentos y junto con un identificador del servidor remoto empaquetarlos en un mensaje; convertirlos a una representación externa de datos y enviarlos al servidor (proceso conocido como aplanado de datos*)*. Luego esperar por la respuesta que se encuentra en una representación de datos externa y convertirla en la representación de datos interna, o desaplanar[7] los datos y devolver los resultados (proceso conocido como "unmarshalling").

En el caso de que esté disponible más de un ORB, deben existir diferentes cabos, cada uno correspondiente a cada uno de los ORBs presentes; y en este caso, es necesario que tanto el ORB como el mapeo al lenguaje, cooperen para asociar los cabos correctos a cada referencia en particular.

4.2.6 Esqueletos de la implantación del objeto

Los esqueletos realizan una función similar a la de los cabos en los clientes, pues es responsabilidad de ellos realizar los procesos de aplanado de datos de los resultados que serán enviados al cliente y de desaplanado de los argumentos necesarios para llevar a cabo la ejecución de algún método, recibidos del cliente. Además, los esqueletos poseen un despachador que es el encargado de llamar al código que implementa el método que el cliente está tratando de invocar y recibir los datos que se obtienen como resultado de dicha ejecución, es decir, el esqueleto junto con el adaptador de objetos coordinan la invocación de los métodos del objeto, observe la figura 4.4.

Para el mapeo a un lenguaje en particular debe existir una interfaz hacia los métodos que implementa cada objeto. Esta interfaz es la que proporciona el acceso a los métodos que habrá de invocar el ORB a través de los esqueletos. Es importante resaltar que la existencia de los esqueletos no implica la existencia de los cabos correspondientes en el cliente, esto debido a que pueden utilizar la interfaz dinámica para realizar la invocación.

[6] Aplanar es el proceso de tomar una estructura de datos y convertirla en un flujo de bytes.
[7] Desaplanar es el proceso de recuperar una estructura de datos a partir de un flujo de bytes.

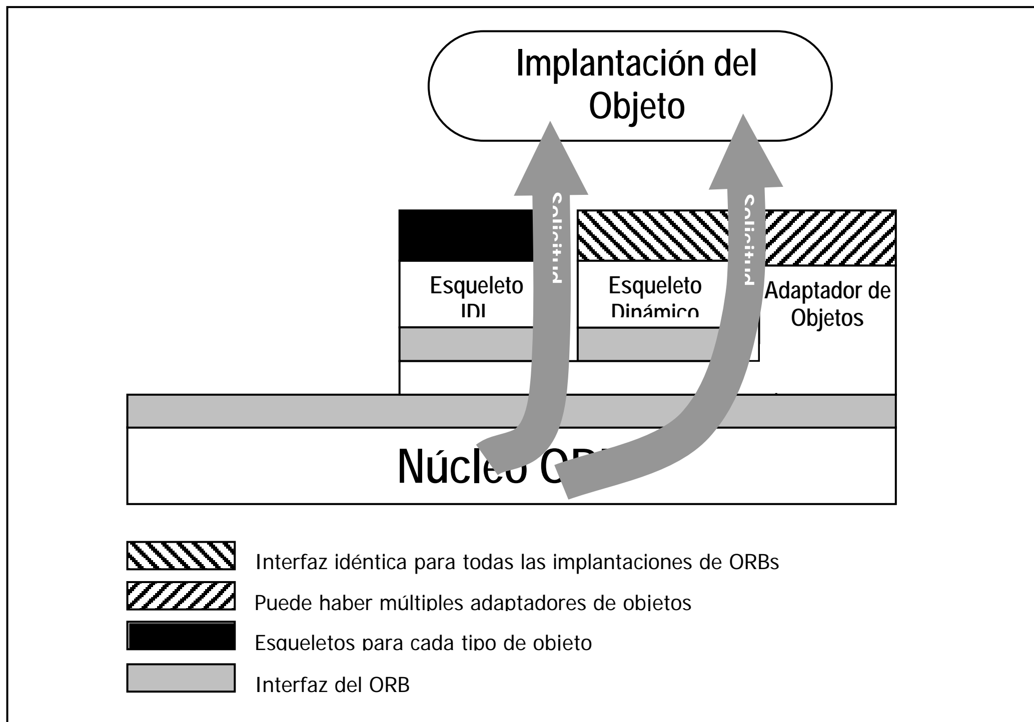


Figura 4.4. Esqueletos CORBA.

4.3 Adaptadores de objetos

El adaptador de objetos es el medio principal por el cual las implantaciones de los objetos pueden acceder a los servicios que proporciona el ORB. Se espera que estén disponibles únicamente unos pocos adaptadores de objetos diferentes, cada uno con interfaces apropiadas para un tipo específico de objeto. Los servicios que proporciona el ORB por medio del adaptador de objetos, generalmente incluyen la generación e interpretación de las referencias a los objetos, la invocación de los métodos del objeto, el proporcionar interacciones seguras, la activación y desactivación de las implantaciones de los objetos, el mapeo de referencias de los objetos hacia sus respectivas implantaciones y, finalmente, el registro de las implantaciones, cuando estas son dadas de alta en el sistema.

La gran variedad de granularidades que pueden presentar los objetos, así como los diferentes tiempos de vida, políticas, estilos en su implantación y otras propiedades, hacen que sea difícil para el núcleo del ORB presentar una interfaz única que sea conveniente y eficiente para todos los tipos de objetos.

Debido a la existencia de esta gran variedad de tipos de objetos con tan diferentes necesidades, se utilizan los adaptadores de objetos para hacer posible que el ORB atienda a grupos de implantaciones de objetos que tengan requerimientos similares de una manera aceptable.

4.3.1 Interfaz del ORB

La interfaz del ORB se encuentra directamente sobre él, es la misma para todos los ORBs y no depende de la interfaz del objeto o del adaptador de objetos que se esté utilizando. Debido a que la mayoría de las funcionalidades del ORB se proporcionan mediante el adaptador de objetos, los cabos, los esqueletos y la invocación dinámica, sólo existen unas cuantas operaciones que son comunes a todos los objetos y que son implementadas directamente por el núcleo del ORB.

4.4 Repositorio de interfaces

El repositorio de interfaces es un servicio que provee objetos persistentes que ponen la información del IDL disponible a tiempo de ejecución. La información contenida en el repositorio de interfaces puede ser usada por el ORB para realizar invocaciones; más aún, utilizando la información contenida en el repositorio de interfaces, es posible que un programa encuentre objetos cuya interfaz no era conocida en el momento de ser compilado, determinar que operaciones están disponibles sobre ese objeto y realizar invocaciones sobre él.

El repositorio de interfaces es el lugar adecuado para guardar información adicional sobre las interfaces de los objetos ORB, tal como, información de depuración, bibliotecas que contengan cabos o esqueletos, rutinas capaces de formatear o desplegar clases particulares de objetos, etc.

4.4.1 Repositorio de implantaciones

El repositorio de implantaciones contiene la información que permite al ORB localizar y activar a las implantaciones de los objetos. La mayor parte de esta información es específica del ORB o del ambiente operativo. Generalmente, mediante operaciones implementadas por el repositorio de implantaciones, se realiza la instalación de las implantaciones y de las políticas de control relacionadas con la activación y ejecución de las implantaciones de los objetos.

El repositorio de implantaciones es el lugar adecuado para guardar información adicional asociada con las implantaciones de los objetos ORB, tales como, información de depuración, de control administrativo, de localización de recursos y de seguridad.

4.4.2 Estructura de un cliente

Un cliente de un objeto es cualquier instancia de software que posee una referencia de él. Una referencia es un elemento que puede ser invocado o pasado como parámetro en la invocación de algún otro objeto. Invocar un objeto involucra especificar el objeto destino, la operación a ser desarrollada, los parámetros que serán pasados al objeto y los que serán devueltos por él.

El ORB maneja las transferencias de datos y de control hacia las implantaciones de los objetos y de vuelta a los clientes. En el caso en que un ORB no pueda completar una

invocación, se devuelve una excepción como respuesta. Para realizar una invocación, el cliente llama a una rutina que se encuentra en su programa que es la encargada de realizar las invocaciones y termina cuando la operación se ha completado.

Los clientes acceden a cabos particulares de cada tipo de objeto, como si fueran rutinas de biblioteca que se encuentren en su programa y por lo tanto dentro de su espacio de direcciones; de esta manera, el programa cliente puede acceder a estas rutinas de manera normal. Todas las implantaciones proporcionan un tipo de dato específico al lenguaje, comúnmente un puntero opaco, para usarse como referencia a un objeto. Para iniciar una invocación, el cliente pasa la referencia del objeto a la rutina de cabo que tiene acceso a la representación interna de las referencias a los objetos e interactúa con el ORB para realizar la invocación.

La forma más común en que un cliente obtiene referencias de objetos, es recibiéndolas como parámetros de salida en invocaciones a otros objetos de los cuales posee una referencia. Cuando un cliente es también una implantación, recibe referencias de objetos como parámetros de entrada en invocaciones a objetos a los cuales implementa. Una referencia también puede ser convertida en una cadena para que pueda ser guardada en un archivo o transmitida por diferentes medios y subsecuentemente puede ser reconvertida por el ORB en una referencia.

4.4.3 Estructura de la implantación de un objeto

La implantación de un objeto provee el estado real del objeto, así como su comportamiento. La implantación del objeto se puede estructurar de muy distintas maneras y además de definir los métodos públicos de la interfaz, generalmente define procedimientos para activar y desactivar objetos y para utilizar otras facilidades proporcionadas por entidades de software que pueden ser o no objetos; esto con el fin de hacer persistente el estado del objeto, para controlar el acceso al objeto o para implantar los métodos mismos.

La implantación de un objeto interactúa con el ORB en muy distintas formas para establecer su identidad, para crear nuevos objetos, y para obtener servicios específicos del ORB. El medio principal del que se vale la implantación del objeto para acceder a estos servicios es el adaptador de objetos que provee una interfaz a los servicios del ORB que es conveniente para cada tipo particular de implantación.

Cuando ocurre una invocación, el núcleo del ORB, el adaptador de objetos y los esqueletos se encargan de que la llamada sea hecha sobre el método apropiado de la implantación. Un parámetro del método, específica el objeto que será invocado y qué método puede usarse para localizar datos del objeto; los parámetros adicionales se proporcionan de acuerdo con la definición del esqueleto, y cuando se completa el método, se transmiten parámetros de salida o excepciones de regreso al cliente.

Cuando se crea un nuevo objeto, se debe notificar al ORB, de modo que tenga conocimiento sobre donde encontrar la implantación de dicho objeto. Normalmente, la implantación se registra a sí misma como la instancia que implementa a un objeto para una interfaz en particular y especifica como iniciar la implantación en el caso de que no se encuentre corriendo.

4.4.4 Estructura de un adaptador de objetos

Un adaptador de objetos es el medio principal mediante el cual implantaciones de objetos acceden a servicios del ORB, como es el caso de la generación de referencias de objetos. El adaptador de objetos exporta una interfaz pública a la implantación del objeto y una interfaz privada a sus esqueletos. El adaptador de objetos es construido sobre una interfaz privada y dependiente del ORB.

El adaptador de objetos es responsable de las siguientes funciones:

- Generación e interpretación de referencias de objetos.
- Invocación de métodos.
- Seguridad de las interacciones.
- Activación y desactivación de objetos e implantaciones.
- Mapeo de referencias de objetos a su correspondiente implantación del objeto.
- Registro de las implantaciones.

Estas funciones son desarrolladas valiéndose del núcleo del ORB y cualquier componente de software que fuese necesario. Es común que el adaptador de objetos conserve su propio estado para acompañar sus tareas, y también es posible que algún adaptador del objeto en particular delegue una o más de sus responsabilidades al núcleo del ORB sobre el cual está construido. El adaptador de objetos está involucrado de manera implícita en la invocación de los métodos, a pesar de la interfaz directa a través de los esqueletos.

Con la ayuda de los adaptadores de objetos, es posible que una implantación de un objeto tenga acceso a servicios sin importar si el núcleo del ORB los implementa o no, en el caso en que los implemente el adaptador simplemente proporciona una interfaz para acceder a ellos; en el caso en que no, el adaptador podría implantarlos por encima del núcleo del ORB. Cada instancia de un adaptador en particular debe proporcionar la misma interfaz y los mismos servicios para todos los ORBs sobre los que esté implantado.

4.5 Protocolo IIOP y GIOP

El GIOP es un protocolo abstracto y el protocolo IIOP (Internet-Inter ORB Protocol) es la implementación del protocolo abstracto GIOP (General Inter-ORB Protocol). La especificación GIOP proporciona un ambiente de trabajo general para protocolos que están construidos arriba de específicas capas transporte. Por lo tanto el IIOP es la especialización de GIOP la cual es construida en la parte alta del TCP/IP [Orfali-G].

El objetivo del protocolo IIOP (Internet-Inter ORB Protocol) es estandarizar la forma de comunicar dos ORB's (pueden ser de diferentes fabricantes) utilizando como base una red TCP/IP. Todo ORB conforme a CORBA debe implementarlo o proporcionar un puente a él. Este protocolo especifica un conjunto de mensajes que heredan de TCP/IP, Representaciones de datos comunes y referencias a objetos Inter-Operable.

En general, la especificación IIOP tiene tres elementos principales [Orfali]:

4.5.1 Requisitos de la administración del transporte.

Dan una vista de la semántica de inicio y finalización de conexiones. El papel de cliente, servidor y de funciones respectivas de cada uno son demarcadas en este nivel. El protocolo descrito es orientado a conexión con bien definidos los papeles para el cliente y servidor.

4.5.2 Definición de código CDR.

El segundo elemento de la especificación IIOP es la representación de datos comunes (CDR por su sigla en inglés). Esta sintaxis de transferencia especifica una codificación para todos los tipos IDL: incluyendo los tipos básicos, tipos estructurados, referencias a objetos (en la forma de IORs) y tipos de pseudo objetos como los TypeCodes. La codificación CDR traduce los tipos IDL en una serie de bytes para hacer un flujo de 8 bytes.

Una característica de CDR es la habilidad para ocuparse de las diferentes tipos de clasificaciones de octetos requerida por diferentes tipos de hardware. La convención adoptada es: el emisor de un mensaje debe hacerlo usando clasificación de bytes nativos (además de un conjunto de banderas en el encabezado del mensaje para indicar la clasificación usada). El receptor del mensaje está obligado a detectar la clasificación usada y el acarreo de alguna conversión, siempre y cuando sea requerida. La ventaja de esta conversiones cuando emisor y receptor ambos usan la misma clasificación de bytes de esta manera no es necesaria la conversión esto eleva la eficiencia.

4.5.3 Formato del Mensaje de IIOP.

El tercer elemento de la especificación IIOP es el formato del mensaje. El protocolo IIOP define siete tipos de formato de mensajes. Los mensajes permiten que los clientes pasen invocaciones hacia los servidores y recibir replicas los cuales pueden ser normal o indicar algún estatus de error. Algunos mensajes adicionales están disponibles para ayudar a manejar la conexión.

Los dos formatos de mensaje más importantes son la petición y la replica. Una operación la cual ha sido declarada en la interfaz IDL para un objeto, es invocada por un cliente usando un mensaje de Petición. El cliente usualmente espera un mensaje de replica desde el servidor la cual contiene normalmente un valor de retorno o posiblemente una condición de error.

Los otros cinco mensajes están enfocados al manejo de algún aspecto de la conexión.

4.6 Aspectos de seguridad en CORBA

Los sistemas distribuidos por naturaleza son más vulnerables que los sistemas tradicionales porque tienen más lugares por donde pueden ser atacados. Cuando se compara los tradicionales sistemas cliente/servidor con los sistemas de objetos distribuidos es algo desafiante, porque los objetos distribuidos puede jugar ambos roles tanto cliente como servidor y esto puede resultar peligroso porque no se tiene un control del comportamiento de la petición.

La seguridad en CORBA es manejada por el servicio de seguridad (“Security Service”) el cual pertenece a los servicios de objetos de CORBA definidos por la OMG. Los

servicios de seguridad definen un marco con funcionalidad para la autenticación, autorización, encriptación y revisión, de esta manera conociendo los requerimientos básicos para la protección de datos sensibles y control de acceso de usuarios a las aplicaciones. Las interfaces definidas son bastantes genéricas, con lo que permiten el uso de diferentes tecnologías de seguridad subyacentes para ser usadas en aplicaciones CORBA. Los Servicios de Seguridad pueden ser implementados para formar uno o dos niveles de seguridad [Orfali-G]:

- Nivel de seguridad 1: es principalmente para aplicaciones las cuales son inconsistentes respecto a la seguridad y por tal motivo tiene requerimientos limitados para hacer cumplir su propia seguridad en términos de control de acceso y auditoria.
- Nivel de seguridad 2: tiene todas las características del nivel de seguridad 1 y también permite aplicaciones para el control de la seguridad proporcionada en la invocación de un objeto e incluir la administración de políticas de seguridad.

La seguridad en CORBA esta basada en los siguientes objetivos:

- Mantener confidencialidad de datos y recursos del sistema.
- Preservar datos e integridad del sistema.
- Mantener responsabilidad.
- Aseguramiento de datos y disponibilidad del sistema.
- Proporcionar seguridad entre sistemas heterogéneos donde diferentes vendedores de ORBs puedan interactuar.
- Proporcionar exclusivamente interfaces de seguridad orientadas a objetos
- Usar encapsulación para promover integridad en el sistema y ocultar la complejidad de los mecanismos de seguridad a través de interfaces simples.
- Permitir implementaciones polimórficas de objetos basadas en diferentes mecanismos subyacentes.
- Asegurarse que la invocación de objetos esta protegida como es especificado por las políticas de seguridad.
- Asegurarse que el control de acceso requerido y la auditoria están ejecutadas a través de invocación de objetos.

Los objetivos antes mencionados, intentan hacer flexible los servicios de seguridad de CORBA. Especialmente el nivel de servicio de seguridad 2, este ofrece aplicaciones para personalizar la implementación de la seguridad. Esta personalización está basada en un objeto de contexto de seguridad llamado *Current,* el cual representa el contexto de ejecución real, en una interacción cliente/servidor, y puede ser usada para obtener ambas credenciales del cliente y servidor. Este contexto es verificado cuando entra una invocación de método en el ambiente del ORB y en la salida de las peticiones del lado donde se encuentra la implementación del objeto. Este contexto de seguridad puede también ser extendido por una seguridad específica que sea necesaria para la aplicación, por ejemplo, proporcionar personalización para la auditoria o control de acceso. Porque el contexto de seguridad relacionado a la invocación de métodos esta implícita o para cambiar la lógica de negocios es mínima.

El control de Acceso de CORBA es verificado por un objeto llamado *AccessDecision,* es la versión mundial para referenciar al concepto de monitor. El objeto *AccessDecision*

proporciona un método llamado access_allowed, este permite determinar si es permitida la invocación del cliente.

4.7 Aspectos de vulnerabilidad del servicio de seguridad de CORBA

A pesar del rico conjunto de características ofrecidas muchas aplicaciones distribuidas basadas en CORBA no toman las ventajas del servicio de seguridad. Las razones más comunes son [Andreas]:

- Carencia de una apropiada implementación del servicio de seguridad en las implementaciones del ORB. Muchos productos CORBA no proporcionan una apropiada implementación del nivel 2 del servicio de seguridad 2.
- Usa solamente cifrado de tráfico de red, muchos productos CORBA ofrecen fácil cifrado de tráfico de red Inter-ORB con IIOP sobre SSL. Esta tiene la ventaja para el desarrollador que usualmente no tiene que hacer algún cambio dentro del código de la aplicación, porque SSL proporciona cifrado transparente para usuarios finales.
- Muchas aplicaciones basadas en CORBA evitan autenticación y autorización por simple tuneleo de credenciales de usuario en los sistemas heredados, el cual proporciona los servicios necesarios para la autorización y control de acceso. Esto es usualmente en el desarrollo de aplicaciones financieras en donde la seguridad es muy importante que sea implementado en los sistemas heredados.

Sin embargo, se desarrollan nuevas aplicaciones haciéndolas más distribuidas. Esto significa que en un futuro se verifiquen todas las operaciones de un sistema. Usando SSL con IIOP debemos tener cuidado de consecuencias importantes de seguridad en el trafico inter-ORB cuando se realizan la invocación de métodos en objetos remotos.

Una de las alternativas en investigación es la implementación del control de acceso en ambientes distribuidos es el uso de certificados digitales, los cuales son sentencias firmadas de acuerdo a las propiedades de las entidades. Porque la flexibilidad ofrecida por los servicios de seguridad de CORBA utilizados en las políticas de control de acceso no resultan ser tan seguras por lo tanto se puede adoptar los certificados digitales para que funcionen como base del control de acceso a CORBA.

5 Arquitecturas de Agentes

5.1 Introducción

5.1.1 Antecedentes

De la misma forma que en la programación OO (Orientada a Objetos) tenemos elementos básicos con los que se trabaja de una forma esencial como son las clases, mensajes, objetos, en la programación de sistemas de agentes tendremos también varios elementos clave. El primero en identificar estos elementos fue Shoham [Shoham] cuando llevó acabo el primer intento de diseñar una programación orientada a agentes. Shoham ve los agentes de la siguiente forma:

"Los agentes resultan entidades computacionales que poseen versiones formales de estados equivalentes a los estados mentales y en particular versiones formales de creencias, habilidades, obligaciones y, posiblemente, otras cualidades mentales"

El lenguaje diseñado por Shoham se denomino Agent0. En la programación usando este lenguaje, un agente consta de las partes que enumeramos a continuación:

- **Creencias:** sentencias lógicas sobre el mundo que conoce, que pueden cambiar en el tiempo.
- **Obligaciones:** sentencias lógicas como las anteriores, que se refieren a la ejecución de acciones que eventualmente se pueden ejecutar.
- **Reglas de obligaciones:** reglas formadas por una condición que implica posibles mensajes a recibir y estados mentales que deben cumplirse cuando se reciben esos mensajes para que la regla se cumpla.
- **Capacidades:** acciones que puede llevar a cabo el agente.

Así, un ejemplo de regla de obligación codificada en Agent0 puede ser el siguiente:

```
(COMMIT
  ( agent, REQUEST, DO(time, action)),
  (B, [now, Friend agent] AND
      CAN(self, action) AND
      NOT [time, CMT(self, anyaction)]),
  self,
  DO (time, action))
```

En la regla anterior podemos ver dos condiciones: la primera tiene que ver con la recepción de un mensaje de tipo REQUEST para realizar una acción concreta en un tiempo determinado; la segunda con la creencia de que el agente emisor del mensaje es amigo, que el agente receptor puede realizar la acción y que además no está obligado a realizar otra acción al mismo tiempo. Una vez que se cumplen ambas, el agente queda

obligado a realizar la acción descrita en el mensaje REQUEST. Este primer lenguaje ha inspirado a muchos trabajos posteriores.

5.1.2 Sistemas Basados en Agentes

Un sistema basado en agentes, es aquel que utiliza el concepto de agente como mecanismo de abstracción, pero aunque sea modelado en términos de agentes podría ser implementado sin ninguna estructura de software correspondiente a éstos, este tipo de sistemas puede contener uno o más agentes.

5.1.3 Web y Agentes Móviles

Fuera del ambiente de la Inteligencia Artificial, existen propuestas de agentes, llamados agentes móviles de Web. Son de diferente naturaleza y son proyectos comerciales más que de investigación. La idea básica es que el agente asiste al usuario en la realización de búsquedas, según parámetros especificados por el usuario, estos realizan solo las tareas de recibir solicitudes, y devolver resultados.

5.1.4 ¿Qué son los Servicios Web?

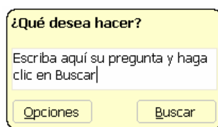
Los Servicios Web son una tecnología emergente impulsada por el deseo de exponer de forma segura la lógica de negocios en Internet. A través de los Servicios Web las empresas pueden encapsular sus procesos de negocios existentes, publicarlos como servicios, suscribirse a otros servicios e intercambiar información entre empresas.

Los principales componentes de los servicios Web son:

- **Servicio**. La aplicación es ofrecida para ser utilizada por solicitantes que llenan los requisitos especificados por el proveedor de servicios.
- **Proveedor de Servicio**. Desde el punto de vista comercial, es quien presta el servicio. Desde el punto de vista de arquitectura, es la plataforma que provee el servicio.
- **Solicitante de servicios**. Desde el punto de vista comercial, la empresa que requiere cierto servicio. Desde el punto de vista de la arquitectura, la aplicación o cliente que busca e invoca un servicio.

5.1.5 Productos o Soluciones Actuales

Existen algunos sistemas que pretenden resolver algunos aspectos como promover la administración de información o conocimiento, manejar grupos de trabajo colaborativo, o trabajar alguno de los aspectos con dispositivos móviles.



El agente de Microsoft es una tecnología de software que permite una forma de interacción con el usuario haciendo que este aprenda a utilizar una computadora de manera más fácil y de una forma más natural, esta tecnología también permite a los desarrolladores crear interfaces conversacionales, incorporando caracteres animados e interactivos en sus aplicaciones y páginas Web.

5.2 Elementos en la Programación de Sistemas de Agentes

5.2.1 Tecnología

5.2.1.1 Topologías: Cliente-Servidor frente a Peer-to-peer

Cliente-Servidor

El modelo cliente-servidor está basado en una clara distinción de actuaciones a seguir entre los nodos denominados cliente y los denominados servidor. Los nodos servidor ofrecen servicios y recursos pero no pueden tomar la iniciativa en la comunicación. Por el contrario, los nodos cliente son los que toman la iniciativa, acceden y usan los servicios que ofrecen los servidores.

Los clientes pueden comunicarse con los servidores pero no pueden comunicarse directamente con otros clientes, y los servidores no pueden comunicarse con los clientes hasta que estos establecen la comunicación. Por esta razón, los clientes deben conocer de la existencia de los servidores pero no necesitan saber nada de otros clientes.

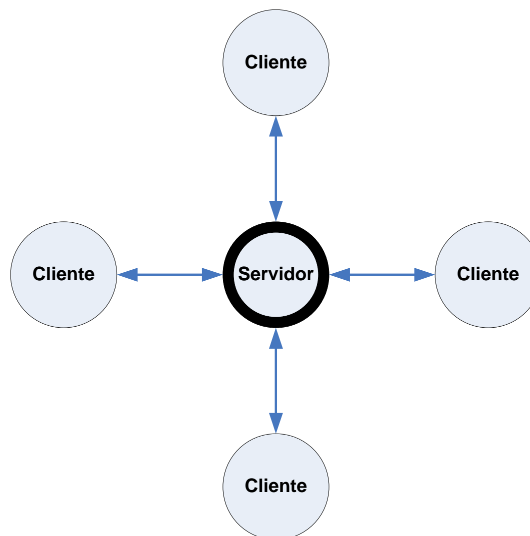


Fig. 5.1. Arquitectura Cliente Servidor.

Peer-to-peer

En el modelo peer-to-peer (P2P), los nodos no toman ningún tipo de tarea en concreto, es decir, un nodo puede actuar de cliente como de servidor. Cada nodo puede iniciar la comunicación con otro nodo, pedir un servicio o ser requerido para un servicio. En el modelo P2P es necesario que los nodos tengan mecanismos para conocer la existencia de otros nodos y de los servicios que ofrecen, esto es lo que se denomina recursos de páginas blancas y páginas amarillas.

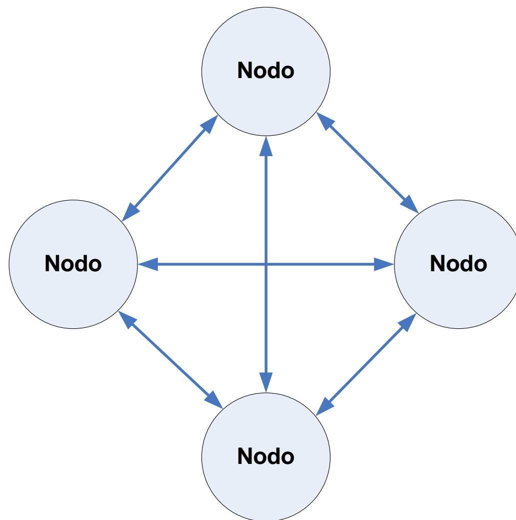


Fig. 5.2 Arquitecturas de redes Peer-to-Peer

Híbridas

En la arquitectura P2P, el hecho de no existir ningún nodo de referencia dificulta la localización de nodos activos. La arquitectura híbrida alivia esta situación ya que debe existir algún nodo *(Súper Nodo)* que proporcione ciertos servicios, y entre ellos el de búsqueda.

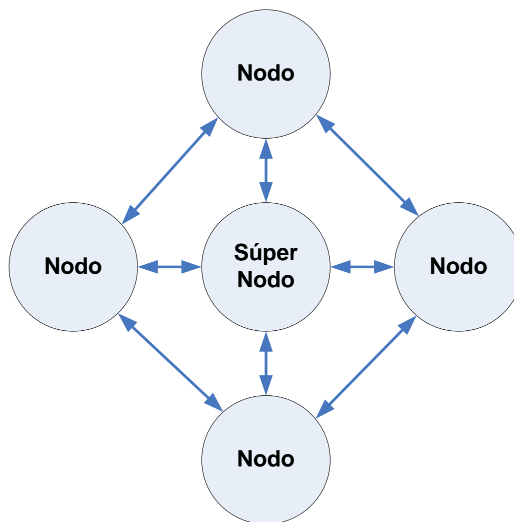


Fig. 5.3 Arquitectura Hibrida

5.2.2 Ontologías en la representación del conocimiento

En estos últimos años se incrementó la cantidad de información en la Internet, lo cual motivó el incremento de la complejidad y la diversidad de la información. Esto, también

motivó el interés de automatizar muchas actividades que tradicionalmente se realizan en forma manual [Baclawski01].

La principal motivación en el uso de las ontologías es que permiten compartir y reutilizar conocimiento en un dominio [Duineveld00]. De ahí el interés de muchos investigadores en el área de inteligencia artificial para tratar de aplicar las ontologías en la representación de conocimiento.

Las ontologías son un elemento importante en la Semántica Web, y es un término filosófico que trata sobre la naturaleza de la existencia y los tipos de objetos que existen [Berners01]. También, la ontología se puede definir como una especificación explícita de una conceptualización compartida [Duineveld00].

La *conceptualización* se refiere a un modelo abstracto de un fenómeno en el mundo, en donde se tienen que identificar los conceptos relevantes de este fenómeno. El término *explícito* significa qué tipos de conceptos son usados y las restricciones de estos conceptos. Y, por último, el término *compartido* se refiere a que las ontologías capturan conocimiento que es aceptado por todos y no es individual de un grupo.

Actualmente, se les está dando más uso a las ontologías en el campo de la Inteligencia Artificial. En este campo se define la ontología como una descripción formal y explícita de conceptos de un dominio (*classes,* a las que nos referiremos como conceptos), propiedades de cada concepto, las cuales describen las características y atributos de los conceptos (*slots* llamados también como propiedades o roles) y restricciones de las propiedades (*facets* conocidas como restricciones) [Fridman01].

5.3 Arquitectura FIPA-JADE.

FIPA (Foundation for Intelligent Physical Agents) es una organización que se encarga de desarrollar especificaciones estándar para los sistemas basados en agentes. Con esto se pretende facilitar la interconexión entre plataformas de diferentes empresas y organizaciones. El objetivo de FIPA es definir la interoperabilidad entre los sistemas basados en agentes, dejando fuera la implementación interna. Define el modelo de una plataforma basada en agentes, el conjunto de servicios que debe proveer y la interfaz entre los servicios.

Para el funcionamiento adecuado de los agentes, se tienen que proveer los servicios y los componentes necesarios para su funcionamiento. Uno de los estándares desarrollados por esta organización es el *FIPA Agent Management Specification* [FIPA02a] en el que se describe el modelo de referencia FIPA para la plataforma de agentes y la funcionalidad de cada uno de sus componentes (ver Figura 5.4).

La plataforma de agentes consiste de los siguientes componentes:

- **Sistema de Administración de Agentes**: controla el acceso y el uso de la plataforma de agentes, se encarga de gestionar el ciclo de vida de los agentes, de los recursos locales y de los canales de comunicación y proporciona el servicio de páginas blancas que permite localizar a un agente por su nombre.

- **Facilitador de Directorios**: provee el servicio de páginas amarillas, que permite localizar a los agentes por sus capacidades y no por su nombre.
- **Canal de Comunicación de Agentes**: se encarga del control de todos los mensajes que se envían entre los agentes, incluyendo los mensajes hacia y desde las plataformas remotas.

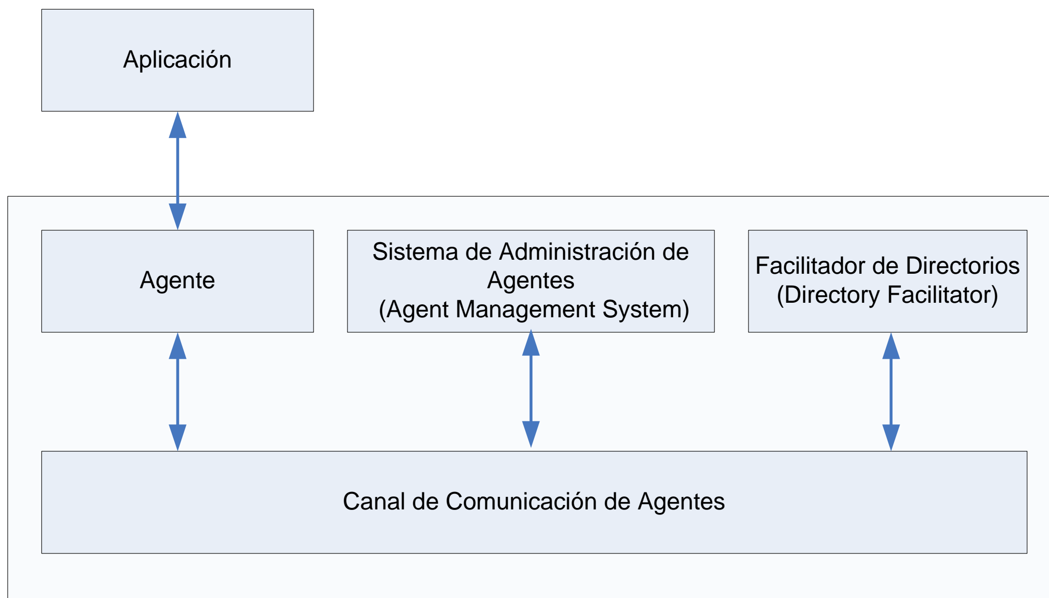


Figura 5.4 Arquitectura de la Plataforma de Agentes Propuesta por FIPA [FIPA02a]

5.3.1 ACL: Una opción de Comunicación entre Agentes

La comunicación es uno de los aspectos importantes dentro de una comunidad de agentes, la cual permitirá compartir y enviar mensajes entre agentes. Los métodos tradicionales permiten la comunicación entre los agentes, pero éstos no son suficientes para lograr un comportamiento social entre los agentes; para ello es necesario que los mensajes tengan un significado y contenido semántico.

La organización internacional FIPA desarrolló un lenguaje estandarizado para la comunicación de agentes denominado *Agent Communication Language FIPA* (ACL FIPA) [FIPA02c], cuyo principal objetivo fue darle un sentido semántico a los mensajes que intercambian los agentes. Para lograrlo se basaron en los *actos comunicativos*, los cuales se basan principalmente en la solicitud de una acción a un agente.

Para la realización de una acción, el agente A envía un mensaje *request* al agente B; el receptor del mensaje podría rechazar o aceptar la acción a través de un mensaje *accept* o *refuse* respectivamente. Si el agente B aceptara la petición tiene que notificarlo e indicarle al agente A cuando finalice la realización de la acción a través del mensaje *agree*. Si en el proceso de realización de una acción se produce un fallo, se notificará al agente A mediante el mensaje *failure*.

En el párrafo anterior se menciona un típico intercambio de mensajes entre dos agentes para la realización de una acción, en donde se hace uso de algunos actos de comunicación como *request* y *refuse*. Éstos son parte importante en la estructura de un mensaje ACL FIPA.

La estructura de un mensaje ACL FIPA está formada por varios parámetros necesarios para la comunicación de los agentes. Un mensaje se compone de un identificador, indicando el tipo del acto comunicativo y también, incluye un conjunto de parámetros de la forma clave-valor que es la información necesaria para la realización de una acción. La estructura del mensaje ACL FIPA [FIPA02c] se muestra en la Figura 5.5

```
(QUERY-IF
 :receiver Buscador@zion:2002/JADE
 :content  <search>
               <cadena>busqueda</cadena>
               <usuario>Cliente@zion:2002/JADE</usuario>
           </search>
 :language XMLCodec
 :ontology ontologiabusqueda )
```

Figura 5.5 Estructura del Mensaje ACL

En la palabra *Quero-If* se especifica el tipo de acto comunicativo que identificará al mensaje que se esté enviando. FIPA identificó varios actos comunicativos [FIPA02b] como: *Accept Proposal, Agree, Cancel, Call for Proposal, Confirm, Failure, Propose, Proxy, Query If, Query Ref, Refuse* y *Request.*

El parámetro *receiver* especifica el agente que recibirá el mensaje. En *content* se especifica el contenido del mensaje. El parámetro *language* contiene el lenguaje con el que se expresará el contenido del parámetro *content*. El parámetro *ontology* especifica el vocabulario que se usará en los mensajes.
3.4 Herramientas para la implementación

5.3.2 IDE (Integrated Development Enviroment)

Un entorno de desarrollo integrado es una herramienta que permite la implementación de proyectos para lenguajes de alto nivel, como por ejemplo Visual C++ de Microsoft o WebSphere de IBM para Java. Estas aplicaciones incluyen potentes editores de texto, gestores de proyectos para el manejo de un gran número de ficheros de código, compiladores para el lenguaje, entornos de depuración y ejecución para la realización de todo tipo de pruebas del software.

Respecto a JACK, esta plataforma incluye su propio IDE que incluye un editor de texto adecuado a las características propias del lenguaje Java extendido, que es usado para la programación de agentes. Además, incluye un compilador para este lenguaje que primero transforma a lenguaje Java el código y posteriormente une el compilador para Java puro produciendo así los bytecodes que ya pueden ser interpretados. En el conjunto de ventanas de compilación del IDE puede encontrarse adjunto un entorno de ejecución

de la aplicación JACK o bien cualquier otra aplicación Java. No existe posibilidad de depurar código con este IDE aunque seria posible, una vez se tiene todo el código Java, utilizar cualquier otro para este propósito.

Zeus es un editor BDI y por lo tanto, en ningún momento se tiene la necesidad de manejar código directamente, aunque realmente el editor BDI este trabajando en la trastienda con un código Java que responde a un modelo de agentes FIPA. Por lo tanto esto va a condicionar decisivamente el que no exista un compilador o depurador de código directamente accesible al programador.

El caso de JADE es distinto al de los dos anteriores. Con JADE debemos programar los agentes usando código 100% Java. Por lo tanto, en el momento de la programación podemos usar cualquier IDE Java que exista en el mercado (o simplemente un editor de texto para proyectos pequeños). Con esto podremos editar, compilar, depurar y ejecutar agentes JADE. Pero siempre trabajando en el nivel de abstracción correspondiente al código Java.

5.4 Arquitectura ADDA

La arquitectura general está planteada en la Fig. 5.6. En esta arquitectura tenemos dos tipos principales de agentes (*AgenteBuscador y AgenteCliente*); uno de estos agentes se encargará de trabajar directamente sobre las peticiones de los clientes o usuarios del sistema, mientras el otro agente se encargará de realizar las búsquedas.

5.4.1 Implementación de los Agentes

Se utilizó Java para el desarrollo de los agentes y las librerías de JADE que proveen las clases necesarias para su construcción. En estos agentes se utilizan dos clases principales que son parte de la librería de JADE: la clase *agent* de la que heredan los agentes y la clase *behaviour* la cual permite definir las tareas que deben realizar los agentes.

AgenteCliente

El *AgenteCliente* se encarga de enviar los mensajes a los agentes del tipo *AgenteBuscador*. El *AgenteCliente* interactúa con el usuario por medio de una interfaz en forma de Applet. El usuario por medio de este tipo de agente puede realizar la acción de búsqueda de información, esta búsqueda será codificada como un mensaje que pueda ser enviado a los agentes del tipo *AgenteBuscador*.

El *AgenteCliente* tiene un método llamando *enviarMensaje* que tiene la finalidad de convertir la búsqueda del cliente en un mensaje que pueda ser enviado al *AgenteBuscador*.

Otro de los métodos es *buscarServicio*, este método se encarga de buscar el servicio de búsqueda, para así saber que agentes proporcionan el servicio de búsqueda de información.

Existe un comportamiento *(Behaviour)* que espera la respuesta de todos los agentes del tipo *AgenteBuscador* que se encuentren activos en ese momento, este comportamiento se llama *RecibeMensajes*, y se repite *n* veces hasta que reciba la respuesta de todos los agentes a los que se les envió la petición de búsqueda. Este comportamiento despliega el resultado de la búsqueda, mostrando la descripción del documento y el enlace a la ubicación donde se encuentra dicho documento.

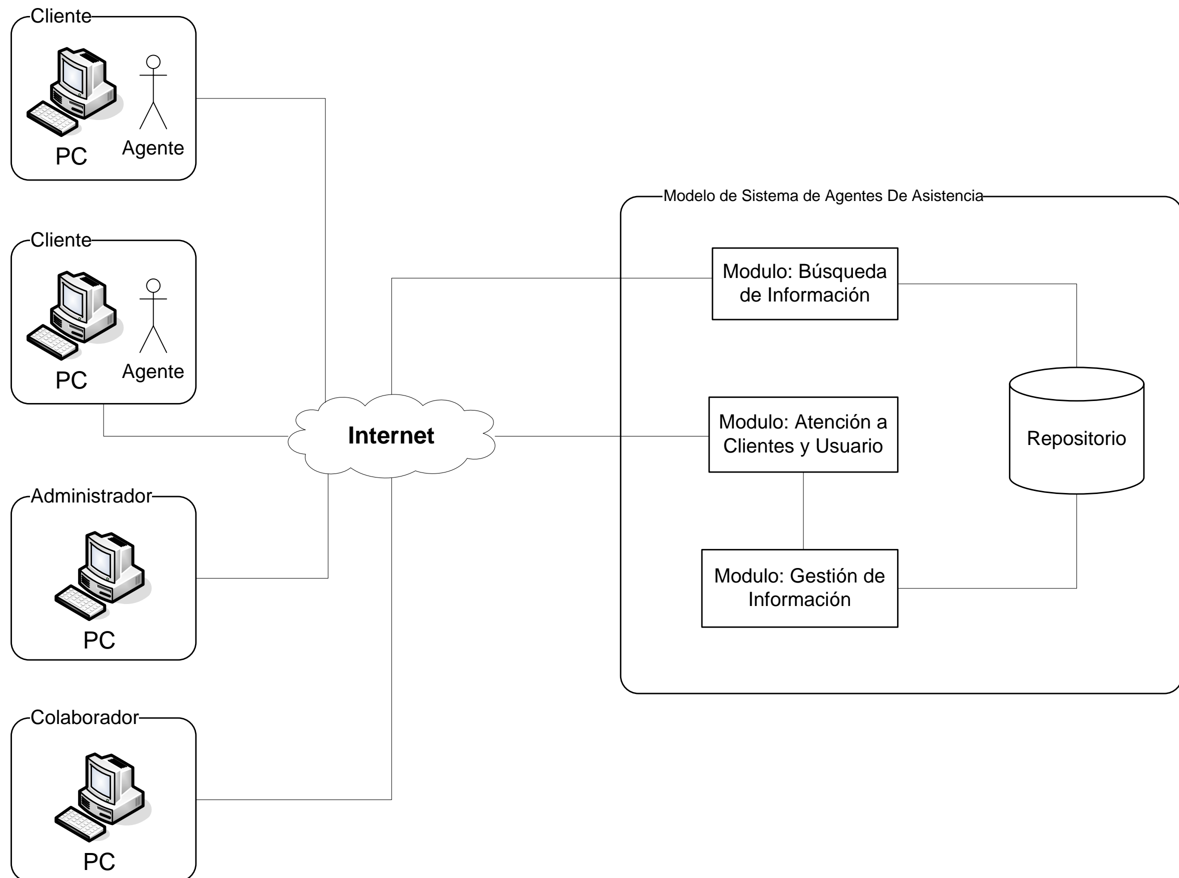


Fig. 5.6. Arquitectura ADDA

AgenteBuscador

El agente buscador se encarga de realizar las búsquedas de uno o más recursos dependiendo del numero de coincidencias que este encuentre, este agente tiene tres comportamientos básicos que conforman su estructura, el primero de ellos es un comportamiento llamando *RecibeMensajes* que tiene la tarea de recibir los mensajes proporcionados por el *AgenteCliente*, otro comportamiento es el denominado *RealizaBusqueda* que se encarga de procesar los mensajes y extraer los parámetros de búsqueda que llegan junto con el mensaje.

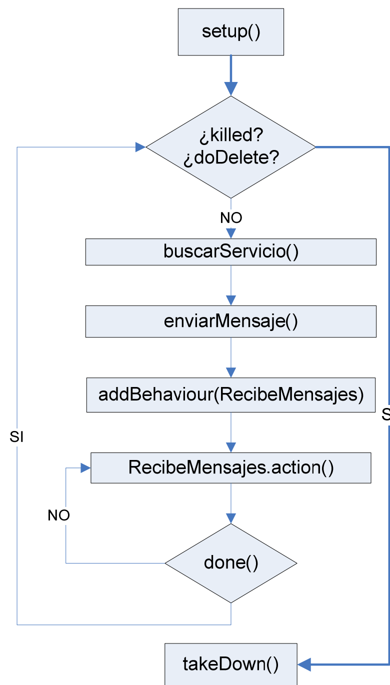
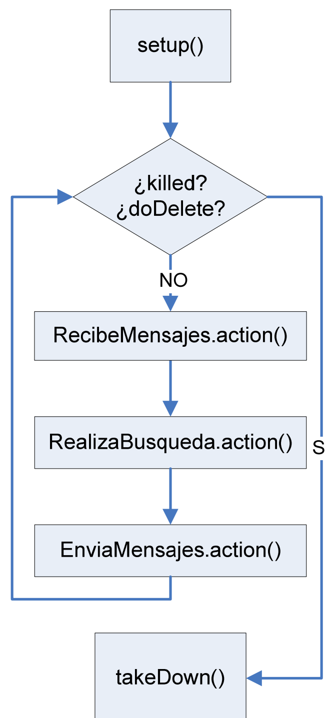


Figura 5.7 Diagrama de ejecución del agente cliente

Finalmente tenemos el comportamiento llamando *EnviaMensajes* este comportamiento procesó el resultado de la búsqueda y lo transforma en un mensaje, que luego es enviado al *AgenteCliente* que solicitó tal búsqueda.



.

Figura 5.8 Diagrama de ejecución del agente buscador

5.4.2 Mensajes Utilizados en la Comunicación entre los Agentes

Para la comunicación de los agentes se usa el lenguaje ACL *(Agent Comunication Language)* cuya estructura está basada en el estándar FIPA-ACL.

Cada uno de estos mensajes está orientado a realizar una acción o a transmitir información de un agente a otro; de esta forma se definen los siguientes tipos de mensajes:

- Mensaje de Petición de Búsqueda. Este tipo de mensaje es enviado a los agentes buscadores para que obtengan determinados recursos definidos en base al criterio de búsqueda.

En este mensaje se utiliza el acto comunicativo *QUERY-IF,* el cual hace referencia a una solicitud de búsqueda de un recurso en la base de conocimiento.

```
(QUERY-IF
 :receiver  agent-identifier: Buscador148.204.183.139@zion:2002/JADE
            agent-identifier: Buscador148.204.183.177@zion:2002/JADE
 :content   <num_resultados>0</num_resultados>
            <search>
               <cadena>busqueda</cadena>
               <usuario>Cliente148.204.183.139@zion:2002/JADE</usuario>
               <busqueda>Description LIKE 'busqueda'</busqueda>
               <codigo>1</codigo>
            </search>
 :language XMLCodec
 :ontology ontologiabusqueda )
```

Figura 5.9 Formato del mensaje de petición de búsqueda

- Mensaje de respuesta de Búsqueda: En este mensaje se indica al agente los resultados arrojados en la realización de la búsqueda de los documentos que estén relacionados al concepto *busqueda*.

En este tipo de mensaje se utiliza el acto comunicativo *INFORM* con el contenido de la búsqueda y una serie de resultados, en caso de que no exista un resultado que coincida con el criterio de búsqueda el número de resultados es 0.

```
(INFORM
 :receiver agent-identifier: Cliente148.204.183.139@zion:2002/JADE
 :content  <num_resultados>2</num_resultados>
           <resultado>
                <descripcion>busqueda</descripcion>
                <link>http://148.204.183.139/busqueda.txt</link>
                <clasificacion>1</clasificacion>
           </resultado>
           <resultado>
                <descripcion>busqueda</descripcion>
                <link>http://148.204.183.177/busquedafiles.doc</link>
                <clasificacion>3</clasificacion>
           </resultado>
 :language XMLCodec
 :ontology ontologiabusqueda )
```

Figura 5.10 Formato del mensaje de respuesta de búsqueda

5.4.3 El Modelo.

El Modelo de Aplicación de la arquitectura es el prototipo de un portal Web desarrollado con tecnologías basadas en Java (JSP's, Servlets) cuya finalidad es mostrar el funcionamiento de la arquitectura ADDA, adicionalmente el modelo provee tareas como son:

- Recopilación, clasificación, recuperación y administración de recursos (documentos)
- Manejo de los recursos en un grupo de trabajo colaborativo.
- Administración del grupo de trabajo
- Acceso al *AgenteCliente* por medio de una interfaz.

En esta sección solo se mostrará el funcionamiento general del modelo de aplicación.

5.4.4 Administración del grupo de trabajo.

La administración del grupo de trabajo se basa en la asignación de tareas especificas a los diferentes elementos de dicho grupo; en el sistema está asignación se da por medio de niveles de acceso al sistema.

Dentro de la lógica de negocio del sistema se definieron tres perfiles *(usuarios, colaboradores y Administradores)* por medio de los cuales se hace una asignación de tareas y recursos a que dichos perfiles tienen derecho.

El siguiente diagrama muestra el funcionamiento general de asignación de tareas entre los diferentes componentes de la arquitectura y el sistema.

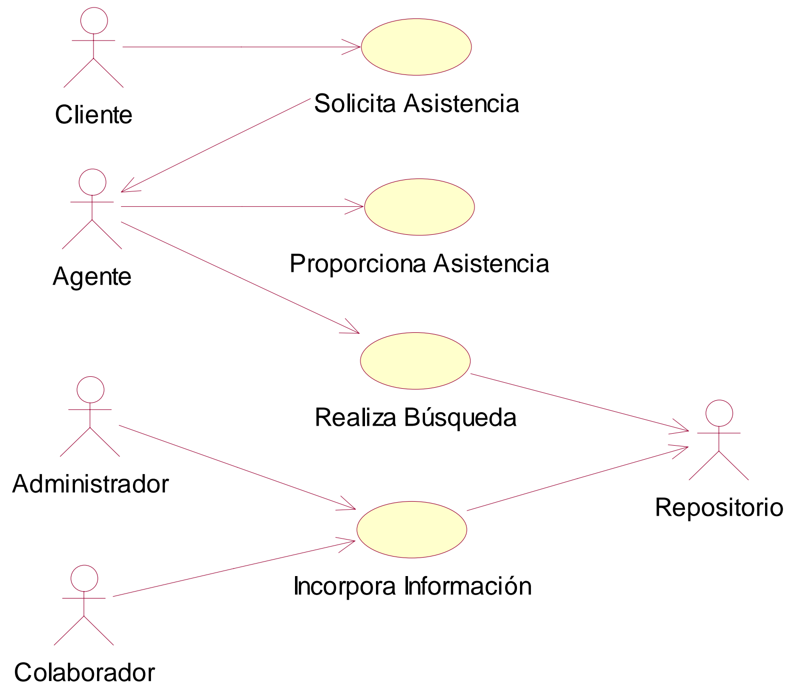


Fig.5.11 "Diagrama General de Casos de Uso".

Administradores ¡¡Bienvenido Admin

| Registro de Usuarios | Incorporar Información | Ver Mis Archivos | Cerrar Sesión | Actualizar Cuenta | Eliminar Cuenta |

Colaboradores ¡¡Bienvenido Colaborador

| Incorporar Información | Ver Mis Archivos | Cerrar Sesión | Actualizar Cuenta | Eliminar Cuenta |

Usuarios ¡¡Bienvenido Usuario

| Cerrar Sesión | Actualizar Cuenta | Eliminar Cuenta |

Fig. 5.12 "Tareas especificas de los diferentes perfiles"

Administración de recursos.

La administración de recursos se apoya en los diferentes perfiles de acceso con que cuenta el modelo de aplicación

Esta administración es la implementación del módulo de Gestión de Información señalado en la arquitectura ADDA, aquí se lleva a cabo toda la administración de la información de la base de conocimientos, esta sección tiene la función de cumplir con las siguientes tareas:

- Incorporación y eliminación de recursos *(documentos)* en el repositorio de conocimientos, por medio de la carga de archivos para ser almacenados.
- Clasificación de los recursos recopilados.
- Realización de anotaciones *(descripción)* para la recuperación de dichos recursos.

6 Referencias

[Redmond] Redmond III, Frank E. DCOM: Microsoft Distributed Component Object Model. Foster City, CA: IDG Books Worldwide, 1997.

[Sessions] Sessions, Roger. COM and DCOM: Microsoft's Vision for Distributed Objects. New York, NY: John Wiley & Sons, 1997.

[Major] Major, Al. COM IDL and Interface Design. Chicago, IL: Wrox Press Inc., 1999.

[Pinnock] Pinnock Jonathan, Professional DCOM Application Development, Wrox Press Ltd.

[Grimes] Grimes Richard, Professional DCOM Programming, Wrox Press Ltd.

[Monson] Richard Monson-Haefel, Enterprise JavaBeans, O'Reilly & Associates.

[Valesky] Tom Valesky, Enterprise JavaBeans(TM): Developing Component-Based Distributed Applications, Addison-Wesley

[Geraghty] Geraghty, S.,COM-CORBA Interoperability, Prentice Hall.

[fciencias] http://www.fciencias.unam.mx/~jloa/SIIGHC/img6.htm

[virusprot] http://www.virusprot.com/Bugs3.html

[iblnews] http://iblnews.com/noticias/10/90785.html

[OMG-OSS] OMG, CORBAservices: Common Object Services Specification, OMG, Noviembre 1996.

[OMG-RT] OMG, Real-Time CORBA, Joint Revised Submission, OMG, Mayo 1999.

[OMG-2.2] OMG, The Common Object Request Broker: Arquitecture and Specification, revision 2.2, OMG, Febrero1998.

[Siegel] Jon Siegel, CORBA 3 Fundamentals and Programming, 2nd Edition OMG Press

[Orfali-G] Robert Orfali, Dan Harkey, Jeri Edwars, The essential Distributed Object Survival Guide, John Wiley & Sons, 1996.

[Orfali] Robert Orfali, Dan Harkey, Jeri Edwars, Instant CORBA, John Wiley & Sons, 1997.

[Orfali-CS] Robert Orfali, Dan Harkey, Jeri Edwars, The esential Client/Server Survival Guide, John Wiley & Sons, 1997.

[Vinoski] Steve Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneus Environments", IEEE Communications, 35(2), pp. 46-55, Febrero 1997.

[Andreas] Vogel Andreas , Bhasker Vasudeven, Maira Benjamin, Ted Villaba, C++ Programming with CORBA, John Wiley & Sons, 1999.

[Garlan] Garlan David, Shaw Mary, An introduction to software architecture, Prentice Hall, 1996.

[Len] Bass Len, Clements Paul, Software architecture in practice, Addison Wesley, 1998.

[vmspec] http://java.sun.com/docs/books/vmspec/

[Shoham] Yoav Shoham. Agent-oriented programming. Art_cial Intelligence, 60(1):51{92, 1993.