Centro de Investigación y de Estudios Avanzados
del Instituto Politécnico Nacional

Unidad Zacatenco

Departamento de Computación

Comparativo de herramientas middleware para el desarrollo de aplicaciones distribuidas bajo el paradigma de la programación orientada a componentes

Tesis que presenta

Ing. Jair Estrada Rosalio

Para Obtener el Grado de

Maestro en Ciencias

en Computación

Director de la Tesis

Dr. José Guadalupe Rodríguez García

México, D.F **Diciembre 2008**

Resumen

La evolución constante de la tecnología y su globalización, ha tenido como consecuencia la aparición de nuevas tecnologías (hardware y software). Además de la necesidad de desarrollar sistemas computacionales distribuidos en el menor tiempo posible (libres de errores, con mejor desempeño y fiabilidad), para ello se ha utilizado el Paradigma de la Programación Orienta a Objetos (POO). Sin embargo, éste último no ha sido suficiente para cubrir los requerimientos de un rápido y mejor desarrollo.

Para cubrir la necesidad de un desarrollo rápido y eficiente, se ha optado por el uso del Paradigma Orientado a Componentes (POC), debido a su cualidad de minimizar en gran medida el tiempo de desarrollo. Sin embargo, para la implementación de un sistema distribuido, es necesario además el uso de herramientas *middleware*, las cuales están especialmente diseñadas para realizar implementaciones robustas de sistemas distribuidos.

La unión del Paradigma Orientado a Componentes y de las herramientas *middleware* ofrece una gran ventaja en el desarrollo de sistemas distribuidos, debido a que los problemas de encapsulación, uso de sistemas heredados, interoperabilidad y heterogeneidad de sistemas operativos y arquitecturas se ven resueltos por estas tecnologías.

En este trabajo hemos establecido un análisis comparativo que se encarga de estudiar tres de las herramientas *middleware* más utilizadas que soportan el uso de la programación orientada a componentes: DCOM, J2EE y CORBA. Cada tecnología ha implementado una especificación particular de componentes: ActivX, EJB y CCM, respectivamente.

Nuestro análisis se encarga de estudiar la filosofía de componentes de cada herramienta y su funcionamiento con respecto a este paradigma de programación. Hemos encontrado un conjunto de similitudes y diferencias en base a características tales como: escalabilidad, maduración, soporte de sistemas heredados y facilidad para la implementación y el desarrollo de sistemas distribuidos, evaluada ésta última característica mediante un caso de estudio implementado con cada una las herramientas estudiadas.

Abstract

The globalization and continuous evolution of technology give as result the need to use new technologies (hardware and software); combined with the need to develop distributed computing systems in the shortest time (bug-free and with better performance and reliability). Today the Object-Oriented Programming (OOP) paradigm has been used to develop this kind of systems, however the OOP fails in meeting the fast and better development requirements.

To meet these needs the Component-Oriented Paradigm (COP) has been proposed thanks its qualities of short development time and reusability. However for implementing distributed systems the *middleware* tools have been proposed giving several mechanisms for distributed systems implementation.

The combination of COP and *middleware* tools offers great advantages for implementing distributed systems given that the problems of encapsulation, use of legacy systems, interoperability and heterogeneity are resolved.

In this work we have carried out a comparative analysis about three of the most popular component oriented *middleware* tools: DCOM, CORBA and J2EE, every one of these tools has his own implementation of components: ActivX, CCM, and EJB respectively.

Our analysis is based on the study of architecture of components in every middleware tool and its operation with respect to the COP. As result we have found several similarities and differences on implementation qualities such as: scalability, maturity, legacy systems support, and facilities on implementing and developing distributed systems, this later quality has been evaluated through a case study implemented with every analyzed tool.

Agradecimientos

A mis padres, Macario Estrada y Rosalba Rosalio
quienes siempre me han dado su amor y el apoyo
necesario para seguir con mis proyectos.

A mis familiares y amigos
quienes siempre me dieron el ánimo
necesario para seguir adelante.

A mi director de tesis
Dr. José Guadalupe Rodríguez García
quien me oriento siempre a lo largo de
mi investigación.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT)
por el respaldo financiero otorgado.

Índice general

Índice de figuras

Lista de Tablas

Capítulo 1

Introducción

Aunque el uso de las herramientas *middleware* prácticamente se ha estandarizado en los últimos años, aun existe la tendencia de realizar mejores desarrollos en el menor tiempo posible, por ello es que el concepto de reutilización cada día toma más auge en este contexto.

Las diferentes herramientas *middleware* existentes han logrado resolver dos de los grandes retos de la comunicación en sistemas distribuidos, interoperabilidad y heterogeneidad de plataformas.

Los retos anteriores han sido resueltos a través de la creación de herramientas *middleware* que funcionan como puentes [37]. Estos puentes son utilizados para permitir comunicaciones transparentes entre plataformas (sistemas operativos y arquitectura de computadoras) diferentes.

Las características de heterogeneidad e interoperabilidad permiten que las comparaciones puedan hacerse entre cualquier clase de sistema operativo y arquitectura de *middleware*.

Si bien en un principio se utilizaban solo herramientas *middleware* que soportaban el paradigma orientado a objetos, ahora se ha optado por la utilización de herramientas *middleware* que soportan el paradigma orientado a componentes. Este cambio de orientación tiene su base en el grado de reusabilidad que los componentes ofrecen en el desarrollo de sistemas.

La combinación de un *middleware* con el Paradigma Orientado a Objetos (POO) se le llama Modelo de Objetos Distribuidos, bajo el mismo sentido, cuando se combina el Paradigma Orientado a Componentes (POC) se llama Modelo de Componentes Distribuidos [14].

Nuestro trabajo está orientado al análisis de las diferentes características inherentes y equivalentes en la utilización de herramientas *middleware* como EJB, CORBA y

DCOM, dentro del dominio del paradigma de programación por componentes para el desarrollo de aplicaciones distribuidas. Esta comparación tiene como resultado un punto de referencia, el cual permite a un desarrollador elegir de forma rápida y sencilla cual de las herramientas antes mencionadas puede utilizar.

Algunos de los criterios de análisis tomados en cuenta son: funcionamiento y escalabilidad, maduración, soporte para utilizar sistemas heredados, existencia de documentación y soporte. Bajo la misma perspectiva se observará un caso de estudio implementado en las diferentes herramientas *middleware*.

Para obtener una comparativa equilibrada se establece una comparación entre herramientas *middleware* que implementan el paradigma orientado a componentes. Con lo anterior se obtiene un análisis que establece criterios para determinar el mejor *middleware* en función del ambiente de desarrollo, objetivos y tipo de sistemas que se pretenden desarrollar.

1.1 Motivación

El desarrollo de aplicaciones distribuidas se ha incrementado considerable en los últimos años. La tendencia actual para el desarrollo de sistemas distribuidos es el uso de herramientas *middleware*, gracias a las facilidades que brindan. Algunos ejemplos de aplicaciones mencionados por Gokhale [16] son: sistemas de reservación de aerolíneas, sistemas de administración de la contabilidad bancaria y sistemas de control de inventarios. Este tipo de sistemas constituyen en la actualidad la mayoría en la tecnología de información mundial.

El desarrollo de sistemas distribuidos puede ser una tarea tediosa y complicada debido a todas las actividades realizadas para lograr obtener el producto deseado. Los sistemas distribuidos pueden integrar componentes heredados, afin de disminuir el tiempo de desarrollo, al brindar mecanismos y facilidades que permiten agilizar el desarrollo, así como garantizar su correcto funcionamiento.

En busqueda de técnicas de desarrollo, herramientas y modelos, que permitieran simplificar el trabajo, a principios de los años 90's se hizo un gran intento por encontrar estos mecanismos y arquitecturas, dando como resultado las herramientas *middleware*. En esos mismos años se elaboraron los estándares que describen la semántica y la estructura de éstos [42].

1.2 Planteamiento del problema

Las investigaciones que se han realizado en torno a comparar las tecnologías concernientes a *middlewares*, han intentado equiparar el Paradigma Orientado a Objetos y el Paradigma Orientado a Componentes. Se necesita establecer un análisis que se enfoque a un solo contexto y que en este caso en particular es el Paradigma Orientado a Componentes.

La comparación se tiene que realizar en herramientas *middleware* Orientadas a Componentes, en donde algunos de los parámetros a evaluar son:

- la escalabilidad,
- la maduración,
- el soporte para utilizar sistemas heredados,
- la facilidad de desarrollo,
- la existencia de documentación.

1.3 Objetivos del proyecto

Objetivo general
Al tener una comparación adeacuada de las diferentes características de las herramientas *middleware* estudiadas, un desarrollador podrá elegir fácilmente que entorno de trabajo se ajusta mejor a sus requerimientos y facilitar el desarrollo de una aplicación distribuida. Para lo cual se tiene que realizar una comparación entre herramientas *middleware* Orientadas a Componentes, estudiando:

- las características de uso,
- el funcionamiento y la implementación,
- el establecimiento de diferencias y similitudes entre ellos,
- la obtención de un marco de referencia,
- el conocimiento de cuales son los casos de uso para los que cada *middleware* podría satisfacer las necesidades del desarrollador,
- la ayuda para elegir el *middleware* adecuado para el desarrollo de aplicaciones distribuidas específicas.

Objetivos particulares

- Comprender el paradigma orientado a componentes.
- Analizar la arquitectura de los *middleware* orientado a componentes elegidos.
- Establecer ventajas y desventajas de cada *middleware* en base a uso, funcionamiento e implementación.
- Implementar un mismo caso de estudio en cada uno de los *middleware* estudiados.
- Evaluar las características de uso, funcionamiento e implementación en el caso de estudio elegido.

1.4 Trabajos Relacionados

La gran relevancia que en los últimos años ha encontrado el uso de *middleware* en el desarrollo de aplicaciones distribuidas ha provocado que se realicen algunos trabajos en torno a comparar su arquitectura, como lo podemos ver por ejemplo en [42], [10] y [30]. Por ello los estudios que se han realizado buscan encontrar similitudes y diferencias para encontrar ventajas y desventajas.

En general las comparaciones realizadas se han hecho entre las siguientes tecnologías OC (Orientado a Componentes) y OO (Orientado a Objetos):

- CORBA (OO),
- DCOM (OC),
- EJB (OC),
- JRMI(OO).

Sin embargo, los análisis que se han realizado bajo el paradigma orientado a objetos (POO), por el gran impacto que ha generado en la industria de desarrollo de software, han tendido a mezclarse con otros paradigmas como es el de componentes.

Las comparaciones que se han hecho intentan equiparar tecnologías de componentes con orientadas a objetos, como se puede apreciar en los trabajos de Juric [22] y de Emmerich [13].

Las investigaciones que se han realizado en torno a comparar las tecnologías concernientes a *middleware*, han intentado equiparar el Paradigma Orientado a Objetos y el Paradigma Orientado a Componentes, generando resultados que se encuentran totalmente desbalanceados debido a la diferencia existente entre paradigmas

Es necesario establecer un estudio que sea equitativo. Por ello hemos establecido un análisis, que versa en torno a un solo contexto, el cual es el Paradigma Orientado a Componentes. El estudio que hemos realizado compara características como son: escalabilidad, funcionamiento, simplicidad, soporte de sistemas heredados, etc, para lo cual se eligieron tres *middleware* que soportan la implementación de componentes: CORBA-CCM, DCOM y J2EE-EJB.

1.5 Organización de la tesis

La estructura de la presente tesis se encuentra organizada de la siguiente forma: en el capítulo 2 se presenta una introducción al paradigma orientado a componentes y herramientas *middleware*, en donde se pretende que el lector obtenga un panorama general en cuanto a componentes y *middleware* se refiere. El capítulo 3 explora las herramientas *middleware* que soportan la implementación de componentes. El capítulo 4 representa la parte medular del trabajo, pues presenta un análisis comparativo de J2EE, CORBA y DCOM, así como la implementación de un caso práctico en cada una de las herramientas mencionadas. En el capítulo 5 se presenta una discusión del trabajo, así como las conclusiones del mismo y el trabajo futuro.

Capítulo 2

Introducción a los componentes y a los middleware

2.1 Introducción al paradigma orientado a componentes

El concepto de componentes de software no es realmente nuevo. Nació a finales de los años 90's [41] como una idea para mejorar la percepción en la reutilización del software. Debido a la problemática surgida en el análisis de la programación orientada a objetos, en donde la reutilización del software no alcanza su máxima expresión.

Para comprender mejor la idea de componente, imaginemos que queremos armar una computadora, para ello tendremos que comprar: una tarjeta madre, un microprocesador, una memoria RAM, un disco duro, una unidad de DVD, un monitor y un gabinete.

Una vez que tenemos cada uno de los elementos con los que podemos *ensamblar* nuestra computadora, ahora solo falta colocar cada pieza en su lugar, cada pieza que se tiene, la conocemos como *componente*, cada pieza funciona de manera independiente, es decir es autónoma y no necesita de algún otro sistema para trabajar.

El componente en sí mismo es un sistema y como tal puede formar parte de otro sistema, formando un microsistema o bien convirtiéndose en un subsistema. Siguiendo con el ejemplo, cada elemento que tenemos para ensamblar nuestra computadora va a interactuar con los demás elementos para obtener una funcionalidad más robusta, por ello en la tarjeta madre ponemos cada pieza en el lugar que le corresponde, mismo que ha sido destinado para cada elemento (figura 2.1), es decir cada dispositivo se va a comunicar con los otros a través de un canal llamado *interfaz*.

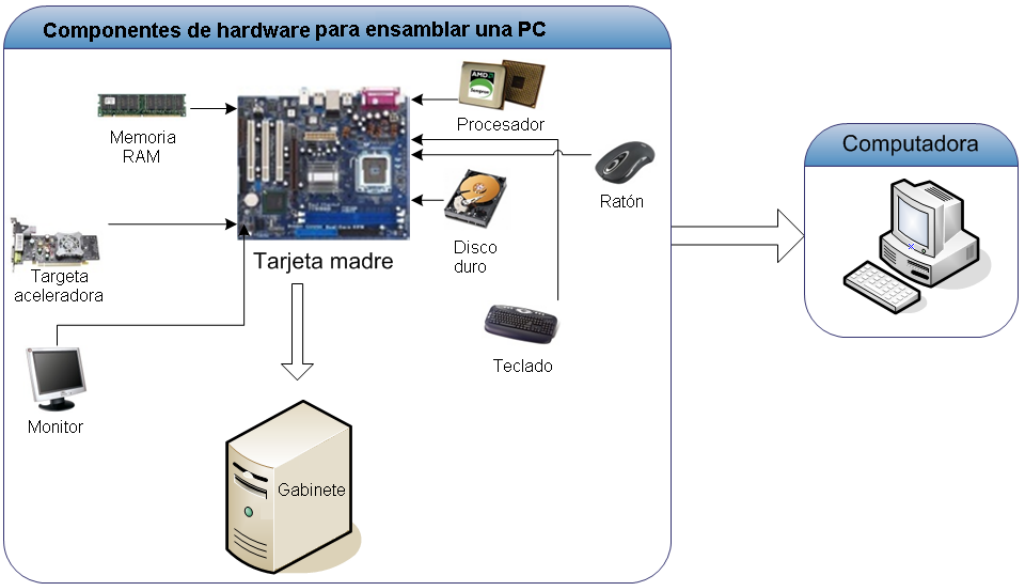

Figura 2.1: Ensamblado de una PC a través de componentes de hardware

2.1.1 Estudio de componentes

En esencia, desde el punto de vista del software, un componente es una pieza de código prediseñado que encapsula alguna funcionalidad expuesta a través de interfaces estándar. Los componentes son los "ingredientes de las aplicaciones", que se juntan y combinan para llevar a cabo una tarea.

El paradigma de ensamblar componentes y escribir código para hacer que estos componentes funcionen se conoce como Desarrollo de Software Basado en Componentes [39]. El uso de este paradigma posee algunas ventajas:

- **La reutilización del software**. Nos lleva a alcanzar un mayor nivel de reutilización de software.
- **La simplificación de las pruebas**. Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados.
- **La simplificación del mantenimiento del sistema**. Cuando existe un acoplamiento débil entre componentes, el desarrollador es libre de actualizar y/o agregar componentes según sea necesario, sin afectar otras partes del sistema.
- **Una mayor calidad del software**. Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

De la misma manera, el optar por comprar componentes de terceros en lugar de desarrollarlos, posee algunas ventajas:

- **Ciclos de desarrollo más cortos**. La adición de una pieza dada, dotada de funcionalidad tomará días en lugar de meses ó años.
- **Funcionalidad mejorada**. Para usar un componente que contenga una pieza de funcionalidad, solo se necesita entender su naturaleza (es decir su interfaz), más no sus detalles internos.

2.1.2 Concepto de componente

Con la idea anterior tratamos de explicar lo que es un componente. Existen varias definiciones al respecto, sin embargo aun no existe un estándar para concretar este concepto. A continuación mencionamos algunas de las definiciones más generales:

- Es una pieza de código auto-contenida y auto-implementable, con funciones bien definidas, la pieza puede ser ensamblada con otros componentes a través de su interfaz [41].
- Una parte reemplazable, casi independiente y no trivial de un sistema que cumple una función clara en el contexto de una arquitectura bien definida [31].
- Es una unidad de composición con interfases especificadas contractualmente y dependencias de contexto explícitas. Un componente de software puede ser implementado de forma independiente y está sujeto a la composición por terceras partes [38]

De las definiciones anteriores podemos inferir que un componente es:
Una pieza de código independiente, auto-contenida y auto-implementable, con funciones bien definidas y a su vez puede ser compuesta de otros componentes. En la figura 2.2 se observa la manera en que se representa a un componente [41] .

2.1.3 Diferencias entre los paradigmas de Compones y Objetos

La Programación Orientada a Objetos (POO) soporta la encapsulación, la herencia y el polimorfismo. Sin embargo nunca llegá a su objetivo porque la herencia viola la verdadera implementación de la encapsulación [41]. Además, los objetos o las clases no son implementados libremente. Podríamos distinguir la programación orientada a componentes (POC) de la programación orientada a objetos(POO) en función de diversos aspectos dentro de la ingeniería de software [41]:

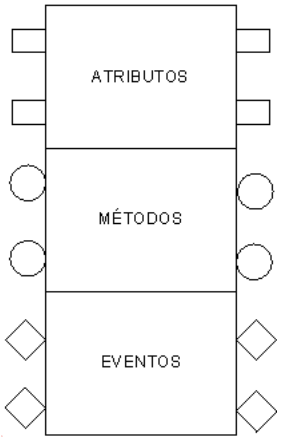

Figura 2.2: Representación de un componente

- La POC está basada en interfaces, mientras que la POO se basa en objetos.
- POC es una tecnología de empaquetado y distribución mientras que POO es una implementación tecnológica.
- POC soporta un alto nivel de reutilización, mientras que POO soporta un bajo nivel de reutilización.
- POC en principio se puede programar en cualquier lenguaje mientras que la POO forzosamente necesita un lenguaje Orientado a Objetos.
- La POC provee un mayor soporte para servicios de orden superior (por ejemplo: composición, portabilidad, etc), mientras que la POO se ha limitado a soportar un pequeño conjunto de servicios tales como seguridad, transacción y así sucesivamente.
- En la POC los componentes están débilmente acoplados, mientras que la POO tiene objetos fuertemente acoplados debido al grado de dependencia que puede llegar a existir entre ellos cuando existe herencia.
- La POC tiene un alto grado de granularidad frente a los componentes, mientras que la POO tiene a los objetos como unidades de composición de grano fino.
- La POC soporta múltiples interfaces y diseños orientados a interfaces, mientras que la POO no provee relaciones transparentes entre las interfaces de superclases y subclases.
- La POC tiene un mayor número de formas de realizar el enlace y descubrimiento dinámico, mientras que la POO provee soporte limitado para la recuperación de objetos y para el soporte de rutinas de composición.

- Los mecanismos de composición por terceros son mejores en la POC, mientras que la POO se ha limitado por tener solo conexiones entre partes.
- La POC ha sido diseñada para obedecer las normas de las plataformas diseñadas por medio de capas de componentes, mientas que en la POO, los objetos obedecen al principio de diseño orientado a objetos.

De lo anterior podemos distinguir a los objetos de los componentes debido a que los segundos obedecen a las reglas de la plataforma para la cual se desarrollan y los primeros están diseñados únicamente para obedecer los principios orientados a objetos. Ademas de que los componentes alcanzan un mayor grado de robustez frente a los objetos tal y como se menciono anteriormente.

2.1.4 Desarrollo de aplicaciones distribuidas orientadas a componentes

La tecnología orientada a componentes permite el uso de software que es desarrollado por diferentes organizaciones, en cualquier lenguaje de programación (por ejemplo: Java, C++, Visual Basic, etc) y ejecutarlo sobre diversos sistemas operativos (por ejemplo: Linux, Windows, MacOs, etc). Los datos son encapsulados en componentes y pueden estar almacenados en cualquier parte, ya sea en una misma computadora o en varias computadoras que se comunican a través de una red.

El párrafo anterior ayuda a comprender un poco la tecnología orientada a componentes, misma que Matjaz y Rozman [22] tratan de mostrar en su trabajo, mencionan que el Modelo de Objetos Distribuidos (establecido en el paradigma orientado a objetos) es la base del Modelo basado en Componentes.

Las aplicaciones distribuidas orientadas a componentes deben de tener las siguientes características:

- funcionamiento del sistema,
- escalabilidad del sistema,
- maduración del sistema, y
- soporte para utilizar sistemas heredados.

En el desarrollo de aplicaciones distribuidas orientadas a componentes, se debe considerar que existen algunas limitaciones [16], tales como:

- **La falta de límites funcionales**: las interfaces son un contrato entre un cliente y un servidor, es decir que no proporciona suficientes mecanismos de implementación para impedir la colaboración de objetos fuertemente acoplados.

- **La falta de aplicación de servidores genéricos**: se tiene que implementar un servidor especial para cada aplicación en donde la complejidad del software aumenta y reduce las características de reusabilidad y flexibilidad de las aplicaciones.

La utilización de componentes extiende los límites de los *middleware* orientados a componentes con interfaces bien definidas así como la composición y ejecución de componentes en servidores de aplicaciones genéricas.

El diseño de software se desarrolla mediante el uso de componentes que interactúan entre ellos para realizar una nueva función. Este tipo de desarrollo permite tener un alto grado de mantenimiento, reutilización, facilidad de actualización y reconfiguración dinámica [24].

Aunado a las características anteriores, es importante tomar en cuenta como se encuentra constituida la arquitectura de cada tecnología, como es que se realiza la administración de las transacciones distribuidas y la manera en que se realiza la comunicación entre los diferentes componentes; así como la configuración a utilizar de acuerdo al modelado basado en componentes, la representación de las entidades y el establecimiento de las conexiones existentes.

Las características antes mencionadas han sido generalmente utilizadas para realizar las diferentes comparaciones que se han realizado a lo largo de estos años acerca de las diversas tecnologías y arquitecturas existentes en el campo de los *middleware*.

2.1.5 Ingeniería de software basada en componentes

La ingeniería de software basada en componentes (ISBC) surge como respuesta a la gran demanda que la idea de reutilización de código generada en el mundo de los programadores a mediados de los años 90's. Se trata de explotar al máximo este concepto debido a que la programación orientada a objetos no lo logró, debido a la limitación que cada desarrollo tiene al depender del lenguaje de programación en donde éste se realiza.

Según Brown [7] la ISBC ha generado un tremendo interés no solo en la comunidad del software sino que también en numerosos sectores de la industria. Los recientes avances tecnológicos como son la Web 2.0, J2EE, COM+ y otros más han estimulado su interés.

La ISBC es el proceso de definir, implementar e integrar sistemas basados en computadora que utilizan componentes de software reutilizables, mismos que son independientes y débilmente acoplados [38].

El uso de la ISBC ayuda a implementar más rápidamente un sistema computacional, en donde se utilizan componentes que ya han sido desarrollados con anterioridad [15], es decir que la reutilización mejora el tiempo de desarrollo e implementación. A este respecto decimos que es mejor reutilizar y/o adaptar que reimplementar.

Existe una clasificación [38] de los fundamentos por los cuales se ha llegado a considerar a la ISBC como parte de las metodologías del ciclo de desarrollo de un sistema:

- **Componentes independientes**: los componentes son completamente especificados a través de sus interfaces. Es decir que existe una clara separación e independencia entre su interfaz y su implementación. Con ésto se evita que cuando existan reemplazos de componentes el sistema pierda su función original.

- **Estándares de componentes**: la definición de estandares facilita la integración de nuevos componentes. Esos componentes se incluyen y se definen dentro de un modelo de componentes. En el nivel más bajo se especifica como se comunican las interfaces con otros componentes.

- **Herramienta middleware**: estas herramientas proporcionan soporte para el uso de componentes en sistemas distribuidos, debido a que maneja la comunicación entre éstos.

- **Procesos de desarrollo**: los procesos que estan definidos para el desarrollo de software son fácilmente adaptables al desarrollo basado en componentes.

El mismo autor propone una lista de características que un componente debe cubrir para que pueda ser utilizado por la ISBC:

- **Estandarizado**: esta característica significa que un componente usado en un proceso ISBC esta ajustado a algún modelo estandarizado de componentes que pueden definir interfaces, metadatos, documentación, composición y despliegue de componentes.

- **Independiente**: un componente debe ser independiente, y de esta manera debería ser posible componerlo y desplegarlo sin tener que utilizar otros componentes específicos. En las situaciones en las que el componente necesita servicios proporcionados externamente, éstos deben ser explícitos en una especificación de interfaz.

- **Componible**: para que un componente sea componible, todas las interacciones externas deben tener lugar a través de interfaces definidas públicamente. Además debe proporcionar acceso desde el exterior a la información sobre sí mismo, como por ejemplo a sus métodos y atributos.

- **Desplegable**: para ser desplegable, un componente debe ser independiente y debe ser capaz de funcionar como una entidad autónoma o sobre una plataforma de componentes que implemente el modelo de componentes. Esto normalmente significa que el componente es binario y que no tiene que compilarse antes de ser desplegado.

- **Documentado**: los componentes tienen que estar completamente documentados para que los usuarios potenciales puedan decidir si los componentes satisfacen o no sus necesidades. La sintaxis e idealmente la semántica de todas las interfaces de componentes tienen que ser especificadas.

Tradicionalmente en la ingeniería de software el proceso de desarrollo consiste en una serie de pasos que están relacionados unos con otros: *análisis, diseño, implementación, pruebas e integración*, mientras que en la ISBC las actividades de desarrollo son: *análisis, diseño, suministro y ensamble*.

Como se puede observar el **suministro y ensamble de componentes**, son las actividades que difieren de la ingeniería de software tradicional con respecto a la ingeniería de software basada en componentes.

Así como en la ingeniería de software tradicional existen nociones que son fundamentales, en la ISBC se han identificado ocho principales conceptos [23] que deben de ser tomados en cuenta:

- **Componente**: es un elemento de software que se ajusta a un modelo. Ese elemento puede ser desarrollado y compuesto sin modificación acorde a un estándar de composición.

- **Modelo de componentes**: define la interacción y la composición concreta por medio de un estándar de componentes.

- **Modelo de implementación de componentes**: denota un conjunto dedicado de elementos ejecutables de software que se requieren para apoyar la ejecución de componentes en base al modelo de componentes.

- **Interfaz de componente**: especifica un flujo de dependencias de componentes que implementan los servicios que serán usados por otros componentes.

- **Contrato**: especifica el comportamiento de la composición, en términos de los componentes participantes. Un contrato define las contrae obligaciones para cada participante así como las operaciones que serán instanciadas a través del contrato.

- **Servicio**: es una abstracción de un conjunto de funciones que son designadas para lograr algún fin lógico.
- **Arquitectura**: los componentes son diseñados sobre una arquitectura predefinida, misma que les permite interactuar con otros componentes y/o plataformas.
- **Patrones**: constituyen los caminos tipicos de la reutilización, debido a que modela el problema y propone una solución.

Desde el punto de vista de los componentes, existen dos actividades que se encuentran inmersas en la ISBC: el *Desarrollo Con Reutilización (DCR)* y el *Desarrollo Para la Reutilización (DPR)*.

Para el primer caso (DCR), los componentes encontrados y recuperados son estudiados y analizados. Estas actividades llegan a ser cruciales para la construcción de la aplicación.

Para el segundo caso (DPR), el desarrollo de los componentes sigue las normas de la ingeniería de software basada en componentes. Para cada instancia los componentes proveen dos tipos de interfaces (ver figura 2.2): *(1) interfaz proporcionada*, quien define los servicios públicos que el componente provee y *(2) interfaz requerida*, que especifica el orden de los servicios que requiere el componente para trabajar apropiadamente.

Dentro del DCR existen algunas tareas de ingenieria de software que se deben cumplir para que el desarrollo sea óptimo [31]:

- **Cualificación de componentes**: los requisitos del sistema y la arquitectura definen los componentes que se van a necesitar. Los componentes reutilizables (tanto DCR como de desarrollo propio) se identifican normalmente mediante las características de sus interfaces. Es decir, se describen los servicios que se proporcionan y el medio por el que los consumidores acceden a estos servicios, como parte de la interfaz del componente. Sin embargo la interfaz no proporciona una imagen completa del acople del componente en la arquitectura y en los requisitos.
- **Adaptación de componentes**: en párrafos anteriores se señalo que la arquitectura del software representa los patrones de diseño que están compuestos de componentes (unidades funcionales), conexión y coordinación. Esencialmente la arquitectura define las normas del diseño de todos los componentes, identificando los modos de conexión y coordinación.

  En algunos casos, es posible que los componentes reutilizables actuales no correspondan con las normas del diseño de la arquitectura. Estos componentes

deben de adaptarse para cumplir las necesidades de la arquitectura o descartarse y reemplazarse por otros componentes adecuados.

- **Composición de componentes**: el estilo arquitectónico juega un papel clave en la forma en que los componentes del software se integran para formar un sistema de trabajo. Mediante la identificación de los mecanismos de conexión y coordinación, la arquitectura dicta la composición del producto final.
- **Actualización de componentes**: cuando se implementan sistemas con componentes DCR, la actualización se complica por la imposición de una tercera parte.

Dentro de la ingeniería de procesos, los componentes pueden ser clasificados dentro de las siguientes cinco categorías [41]:

1. **Componente de especificación**: esta forma representa la especificación de una unidad de software que describe el comportamiento de un conjunto de componentes objeto y define una unidad de implementación. El comportamiento es definido como un conjunto de interfaces. Un componente de especificación es realizado como un componente de implementación.
2. **Componente de interfaz**: la interfaz representa la definición de un conjunto de servicios que pueden ser ofrecidos por un componente objeto.
3. **Componente de implementación**: la implementación es la realización del componente de especificación, que es autoejecutable. Lo cual quiere decir que puede ser instalado y remplazado independientemente. Esto no significa que es totalmente independiente de otros componentes, es decir que pueden existir dependencias en cuanto al funcionamiento del sistema, pues éste realiza una tarea especifica.
4. **Componente instalable**: es instalado y ejecutado como una copia del componente de implementación. El componente es ejecutado dentro de la plataforma del sistema en que se esté trabajando, creando una instancia del componente en tiempo de ejecución cuando el sistema operativo detecta que necesita realizar alguna operación.
5. **Componente objeto**: es una instancia de un componente instalable, puede tener múltiples instancias.

2.2 Introducción a los middleware

2.2.1 Introducción

La gran importancia que en los últimos años ha tomado el uso de los *middleware* en el desarrollo de sistemas distribuidos se debe principalmente, a las facilidades que

brindan para el desarollo y la implementación de este tipo de sistemas. En los trabajos de Zarras [42] y Vassilopoulos [10] se pueden encontrar algunos puntos interesantes en torno a las similitudes y diferencias, así como ventajas y desventajas en el uso de estas arquitecturas.

2.2.2 Concepto de middleware

El *middleware* consiste en un conjunto de mecanismos reutilizables que ofrecen soluciones a problemas como por ejemplo, la heterogeneidad, interoperabilidad, seguridad, fiabilidad, etc. Estas funciones son ofrecidas o bien por el núcleo de la infraestructura del *middleware* o por servicios complementarios.

En la literatura de los sistemas distribuidos existen varias definiciones con respecto al término middleware, a continuación citamos algunos de ellos:

- Una capa de software que se encuentra entre el sistema operativo y la aplicación, facilita la integración transparente de objetos distribuidos [42].
- Un software posicionado entre el sistema operativo y la aplicación [32].
- Una capa entre el sistema operativo de red y la aplicación de **componentes**. Resuelve la heterogeneidad y facilita la comunicación y coordinación de componentes distribuidos [13].

Como se observa no existe gran diferencia entre un concepto y otro, en base a lo anterior podemos definir a un *middleware* como:

Una capa de software que se encuentra entre el sistema operativo y la aplicación. Ésta facilita la comunicación, coordinación e integración de componentes u objetos, mediante un conjunto de servicios que pueden ser internos o externos y se encarga de resolver los problemas de heterogeneidad existentes entre sistemas operativos.

Un sistema distribuido requiere de una infraestructura que soporte adecuadamente el desarrollo y la ejecución de aplicaciones distribuidas. Las plataformas *middleware* presentan tales infraestructuras debido a que éstas proveen un canal de comunicación (llamado *buffer*) entre la aplicación y la red.

La red es únicamente un mecanismo de transporte, el acceso a ella depende de la tecnología y de los recursos físicos que se tengan (de antemano se sabe que en la mayoría de los casos las plataformas implantadas inherentemente son heterogéneas).

El objetivo principal de las plataformas *middleware* es hacer homogéneo el acceso a la red y ofrecer servicios genéricos que soporten la ejecución distribuida de las aplicaciones [21], es decir que este tipo de software es un programa que permite realizar la comunicación entre diferentes arquitecturas de red y sistemas (ya sean operativos o de aplicación). En otra palabras es un programa de capa intermedia.

2.2.3 Objetivos del middleware

El *middleware* debe proveer aplicaciones con primitivas de alto nivel que simplifiquen la construcción de sistemas distribuidos.

Los retos a vencer son muchos y muy variados en el desarrollo de sistemas y aplicaciones distribuidas, sobre todo el problema de la heterogeneidad. Por ello existen algunos requerimientos que el *middleware* debe satisfacer para poder cumplir con su tarea. Emmerich [13] establece los siguientes puntos:

- **Apertura**: la infraestructura del *middleware* debe permitir extender la construcción de la aplicación en varios caminos (por ejemplo añadiendo, removiendo o actualizando servicios).
- **Comunicación de red**: es el medio de transporte que utilizan los sistemas distribuidos para comunicarse es una red. Los recursos se encuentran ubicados en diferentes computadoras y los componentes necesitan interactuar unos con otros. Esta interacción solo puede ser alcanzada mediante el uso de protocolos de red.
- **Coordinación**: los componentes del sistema distribuido residen en diferentes nodos, por ello es necesario sincronizar la comunicación e interacción entre ellos para evitar el bloqueo de recursos y asegurar que todos puedan utilizarlos.
- **Escalabilidad**: permitir el crecimiento del sistema, sin alterar ninguna de las tareas que éste realice, es decir, debe soportar la adición de componentes y el cambio de localización sin afectar la infraestructura del sistema.
- **Fiabilidad**: el sistema debe asegurar que los mensajes lleguen completos a su destino. En caso contrario deberá tener mecanismos de replicación que aseguren el correcto envío y recepción de los datos. La infraestructura *middleware* debe ser eficiente y si es necesario prever la ejecución de las aplicaciones que están en la capa superior.
- **Heterogeneidad**: los componentes de sistemas distribuidos, pueden incluir componentes nuevos o hererados, esto resulta por lo general en un conjunto de recursos heterogéneos. Esta heterogeneidad tiene diferentes dimensiones: sistemas operativos, hardware, lenguajes de programación e incluso con el mismo

middleware. Este último debe permitir la interacción de todos los componentes de forma transparente.

- **Soporte del modelo de objetos y componentes**: el *middleware* ofrece mecanismos que incorporan el soporte del modelo de objetos y de componentes.

- **Interacción operativa**: el *middleware* permite que las operaciones de dos componentes interactúen entre sí. La técnica que se usa para satisfacer este requerimiento, es por medio de invocación de métodos.

- **Interacción remota**: el *middleware* permite la interacción entre dos componentes alojados en diferentes lugares (ya sea que éstos se encuentren en directorios o nodos remotos).

- **Independencia tecnológica**: el *middleware* soporta la integración de diferentes arquitecturas tecnológicas.

- **Transparencia de distribución**: desde el punto de vista de la interacción de componentes u objetos de un programa, ésta es idéntica tanto en el caso de una invocación remota como de una local. Esta propiedad indica que la aplicación debe ser percibida por los usuarios o desarrolladores como un conjunto, más que como una colección de elementos independientes.

Con respecto al punto anterior, Zarras [42] propone que la transparencia de distribución es un concepto bastante genérico, por ello es necesario enmarcarlo dentro de un conjunto de características más especificas que permitan comprender a profundidad este concepto:

- **Transparencia de acceso**: la infraestructura debe permitir el acceso local y remoto de los elementos de la aplicación. El mecanismo de acceso debe de ser el mismo.

- **Transparencia de ubicación**: la infraestructura debe permitir el acceso a los elementos de la aplicación sin necesidad de conocer su ubicación física.

- **Transparencia de concurrencia**: la infraestructura debe permitir ejecutar procesos concurrentes sin interrupción alguna.

- **Transparencia de fallas**: la infraestructura debe proveer servicios de control y recuperación de fallos.

- **Transparencia de migración**: la infraestructura debe proveer medios para cambiar de lugar los elementos de la aplicación sin dañar la correcta ejecución de la aplicación. Por lo tanto no se debe afectar a los elementos que se encuentran fuertemente acoplados a los componentes migrados.

- **Transparencia de persistencia**: la infraestructura debe proveer mecanismos para ocultar la activación y desactivación de elementos en el uso de recursos compartidos.
- **Transparencia de transacción**: la infraestructura debe proveer mecanismos para la coordinación de ejecución de transacciones aisladas y atómicas.

2.2.4 Arquitectura de un middleware

En general todas las infraestructuras *middleware* asumen un estilo de arquitectura, misma que es seguida por las aplicaciones. Existen tres principios básicos que se deben tomar en cuenta para soportar este tipo de arquitecturas en el desarrollo de aplicaciones o sistemas *abiertos* y *escalables*:

- **Modularidad**: las aplicaciones deben consistir en una colección de elementos, cada uno provee servicios que son usados por otros componentes. La aplicación permite la identificación de dependencias entre los elementos que conforman el sistema. Consecuentemente éstos permiten determinar qué elementos son afectados por alguna adición, remoción o actualización eventual
- **Encapsulación**: para cada elemento que constituye el sistema, existe una clara separación entre los elementos de la interfaz y la implementación. La interfaz es una especificación bien definida de servicios, existe un contrato entre los elementos y las entidades utilizadas. La implementación es la utilización de los servicios prestados.
- **Herencia**: cualquier interfaz puede ser derivada de otra interfaz. La interfaz derivada contiene los servicios de la interfaz base.

Los elementos que interactúan con la plataforma de un *middleware*, ver figura 2.3, en la mayoría de las veces son heterogéneos y se dividen en:

- **Lenguajes de programación**: diferentes componentes pueden ser desarrollados en diferentes lenguajes de programación.
- **Sistemas operativos**: los sistemas operativos tienen diferentes características y capacidades.
- **Arquitectura de computadoras**: cada computadora tiene una composición diferente, tanto de elementos como del tipo de representación de datos que maneja así como la manera de procesar estos datos.
- **Red**: diferentes computadoras son conectadas entre sí a través de diferentes tecnologías de red.

El *middleware* se encarga de eliminar la heterogeneidad que existe entre cada elemento y facilita la coordinación y comunicación entre componentes ya sea de una aplicación o sistema distribuido.

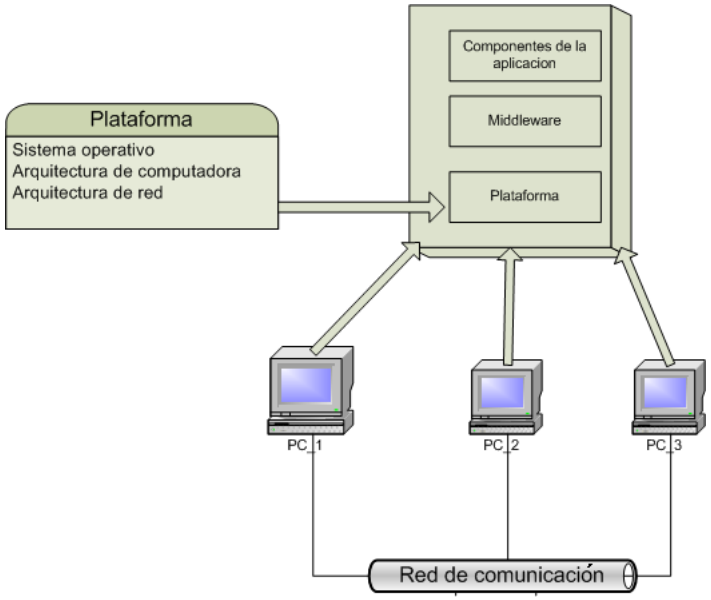

Figura 2.3: Arquitectura de un middleware

2.2.5 Servicios fundamentales del middleware

En la sección anterior observamos que las características de apertura, escalabilidad y transparencia de acceso dependen directamente del tipo de aplicación que se esté realizando. Sin embargo, es importante señalar que en los *middleware* no importa el tipo de sistema de que se trate, el *middleware* debe proveer un conjunto de servicios fundamentales que ayuden a asegurar el buen funcionamiento del sistema que se pretende desarrollar.

Los servicios por lo general se encuentran divididos en tres categorías: *repositorio de servicios, coordinación* y *seguridad* [42]. Para alivio de los diseñadores y programadores ésta es tarea de los proveedores del *middleware*, por lo que los desarrolladores no tendrán que preocuparse por programarlos.

2.2.6 Middleware y los sistemas distribuidos

Un Sistema Distribuido es usualmente definido como:

- Una colección de computadoras independientes que aparecen ante los usuarios del sistema como un único sistema [36],
- Un sistema en donde los componentes se encuentran situados en computadoras diferentes distribuidas a través de una red, estas coordinan y comunican sus acciones únicamente mediante el paso de mensajes [9],

- Un sistema de procesamiento de información que contiene varias computadoras independientes, que cooperan unas con otras mediante una red de comunicacion para lograr un fin común [32].

De la relación entre las definiciones anteriores podemos concluir que un Sistema Distribuido: *es un conjunto de computadoras que comparten recursos que se encuentran distribuidas a través de una red. Estas computadoras se comunican y se coordinan mediante el paso de mensajes para lograr un objetivo común.*

2.2.7 Interoperabilidad y heterogeneidad

Las diferentes arquitecturas de *middleware* existentes han logrado resolver dos de los grandes retos de la comunicación entre sistemas distribuidos, interoperabilidad y heterogeneidad de plataformas.

Las características anteriores han sido resueltas a través de la creación de *middleware* que funcionan como puentes [37]. Estos puentes son utilizados para permitir comunicaciones transparentes entre plataformas (sistemas operativos y arquitectura de computadoras) diferentes.

Las características de heterogeneidad e interoperabilidad permiten que las comparaciones puedan hacerse entre cualquier clase de sistema operativo y arquitectura de *middleware*.

2.2.8 Aspectos de comparación entre middleware

Las comparaciones que se han realizado están basadas en la infraestructura de diferentes herramientas *middleware*. En general se han comparado arquitecturas como CORBA, Web Services, J2EE, DCOM en base a características como tiempo de petición y de respuesta, referencias remotas, interfaz, etc. Comúnmente buscando la integración o comunicación de objetos distribuidos.

El trabajo propuesto por Zarras [42] presenta los tipos de análisis que se han realizado en torno a la infraestructura de los middlewares, en donde se ha estudiado su funcionalidad. Sin embargo menciona que la propuesta de algunos autores no establece relación alguna entre los conceptos funcionales y los patrones típicos de los requerimientos impuestos por las aplicaciones distribuidas.

Establecer una infraestructura que permita satisfacer los requerimientos de una aplicación específica es complicado, debido a la independencia de cada caso, pues cada tecnología se enfrenta a la existencia de ventajas y desventajas que cada arquitectura presenta, es decir, que en general no existe un *middleware* perfecto, sin embargo es perfectible.

Estas características no están muy lejos de las distinguidas por Emmerich [13], en donde por un lado muestra los requerimientos funcionales que debe tener un *middleware*: *comunicación, coordinación, fiabilidad* y *escalabilidad* y por otro menciona las diferentes categorías en que se dividen éstos: *middleware de transacciones, middleware orientado a mensajes, middleware orientado a procedimientos, middleware orientado a objetos* y *middleware orientado a componentes*.

La existencia de diferentes *middleware* no puede asegurar la existencia de una plataforma que permita dar soporte a los requerimientos típicos de un sistema distribuido. Por ello la propuesta de Emmerich [13], es realizar un análisis, que permita evaluar un enfoque, en donde se combina tanto la funcionalidad los *middleware* como los requerimientos de una aplicación distribuida, para ello propone:

- Definir un estilo de arquitectura genérica que satisfaga los requerimientos claves, si el estilo de la arquitectura particular de la infraestructura de un *middleware* se ajusta este estilo de estructura genérica, la infraestructura es capaz de satisfacer los requerimientos clave.
- Identificar servicios fundamentales que deberían ser ofrecidos por la infraestructura de un *middleware* para satisfacer los requerimientos clave.

La comparación de las características que propone las evalúa en tres diferentes *middleware*, presentando un cotejo detallado de CORBA, J2EE y COM+. La conclusión del estudio realizado es que éste permite elegir un *middleware* de los antes mencionados para desarrollar un sistema distribuido en base a los requerimientos de la aplicación. El problema de su estudio radica en la relación de heterogeneidad existente entre las diferentes arquitecturas *middleware*, desde el punto de vista de objetos y componentes.

Por otro lado Vassilopoulos [10] realiza una comparación similar basando su investigación en la evaluación del funcionamiento de tecnologías para el desarrollo de aplicaciones distribuidas.

El objeto de realizar la evaluación del funcionamiento deriva de establecer una estimación del tiempo de procesamiento para cada tecnología. El estudio lo realiza mediante la implementación de dos pruebas, en donde se invocan dos métodos diferentes en un número cualquiera de veces, que permite determinar el tiempo que tarda cada tecnología en realizar su procesamiento.

Capítulo 3

Middleware a base de componentes

3.1 DCOM

Antes de introducir DCOM (Distributed Component Object Model) y COM (Component Object Model), es importante conocer cómo surge esta tecnología, que lejos de intentar ser un paradigma de desarrollo, su intención principal era dar solución al problema de la creación de documentos compuestos, es decir, una especie de contenedores, cuyos componentes son piezas heterogéneas de información, que pueden haber tenido origen en diversas aplicaciones informáticas (un procesador de textos, un gestor de proyectos, una base de datos, una hoja de cálculo, etc).

Para solucionar el problema anterior, nace la tecnología OLE (Object Linking and Embedding) que en español significa *Objetos enlazados y vinculados*. Éstos constituyen un conjunto de servicios unificados, fundados en abstracción de entidades (objetos), con la capacidad de tipificar servicios ofrecidos por otras entidades, ampliando la robustez de cada uno para finalmente obtener un conjunto rico de componentes.

Estos componentes podían aparecer dentro de un documento compuesto, como un acumulado de información, sin embargo este documento posee inherentemente elementos que han sido elaborados por aplicaciones externas (aquí podemos observar como comienza a aparecer el concepto de la reutilización de componentes).

Con la aparición de OLE 1.0, los usuarios fácilmente combinaban información de diferentes fuentes [8]. Propiamente hablando se decía que dentro de una hoja de procesador de textos como Word podía exitir una hoja de cálculo como Excel (ambos paquetes propiedad de Microsft). Con esto se pretendía dar al usuario un conjunto de herramientas que le permitieran formar, aún que éste no lo supiera, un sistema basado en componentes siempre independientes uno del otro que le ayudaría a facilitar su trabajo.

En el desarrollo de la siguiente versión de OLE surgió un nuevo problema, éste era el cómo proporcionar servicios entre componentes. Para ello se crearon un conjunto de técnicas que pretendían dar solución al problema antes planteado. Entre toda la gama de tecnologías que había surgido, COM constituyó la base principal para OLE 2.0, ésta fue COM. De esta manera ya no solo se solucionaba el problema de los documentos compuestos, sino que se pudo dar solución a muchos otros problemas.

Es en esta forma que OLE proporciona un estándar de comunicación entre objetos y componentes. Los objetos y/o componentes no necesitan conocer de forma anticipada con que componente se van comunicar y tampoco existe la necesidad de que el código haya sido escrito en el mismo lenguaje.

Esta característica en gran parte fue el resultado de la introducción de COM. Comienza por establecer un nuevo paradigma, en donde existe la libre interacción entre programas, bibliotecas y aplicaciones, dándo como resultado un mayor potencial y robustez al sistema.

Finalmente OLE (hoy Activex) deja de ser una tecnología de solo documentos compuestos y pasa a ser parte de COM, como una nueva forma de programar sistemas.

3.1.1 Filosofía de COM

La comunicación depende enteramente del componente con que se este trabajando. La figura 3.1 presenta un ejemplo de la tecnología COM, en donde por medio de una biblioteca se acceden a los servicios de una aplicación, que ejecuta un proceso diferente al de la aplicación que está realizando la petición. La comunicación de los procesos que se están ejecutando de manera local se hace por medio de la intercomunicación de procesos.

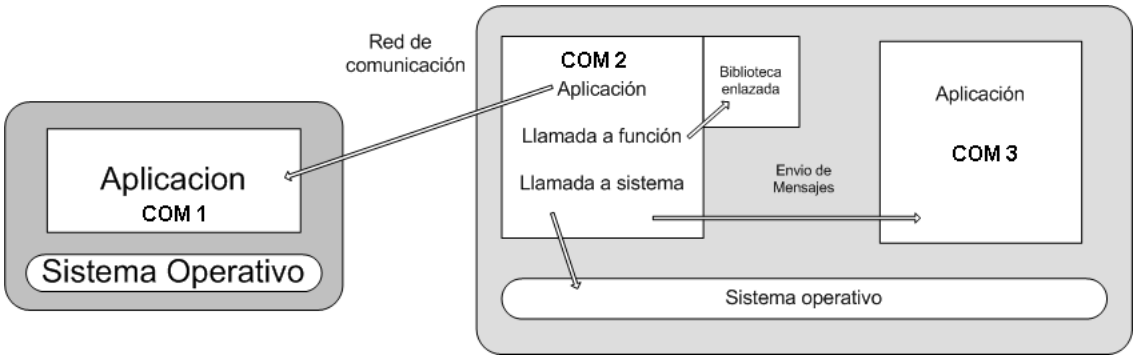

Figura 3.1: Arquitectura COM

La arquitectura COM define un estándar mediante el cual un componente ofrece servicios a otro componente, es decir trabaja mediante el concepto de caja negra. COM aplica una arquitectura de servicio común e independiente de bibliotecas y aplicaciones que transforma la manera en que se implementa un sistema.

3.1.2 Funcionamiento de COM

En COM cada paquete de código implementa sus servicios como componentes. Cada componente proporciona una o varias interfaces, éstas a su vez implementan una serie de métodos [19].

En la filosofía de componentes un método es una función que realiza una determinada tarea, ésta puede ser llamada por algún proceso en ejecución usando una instancia de COM [1]. Como puede apreciarse este concepto no dista mucho del que se define en el paradigma orientado a objetos.

Cualquier proceso que necesite acceder a los servicios proporcionados por un componente lo puede hacer utilizando la interfaz como medio de comunicación. Si se intenta acceder a cualquier método de forma directa no podrá realizarse debido al grado de encapsulación que existe.

En la figura 3.2, se muestra la representación de un componente de tipo COM, en general casi todos los componentes pueden proveer varias interfaces diferentes. En la imagen se muestra un componente que tiene tres interfaces diferentes, mismas que son representadas mediante un circulo que sale de uno de los extremos del componente.

Los componentes son siempre implementados dentro de un contenedor. Este contenedor puede ser una librería de enlace dinámico que es ejecutada cuando algún proceso en ejecución la requiere [12].

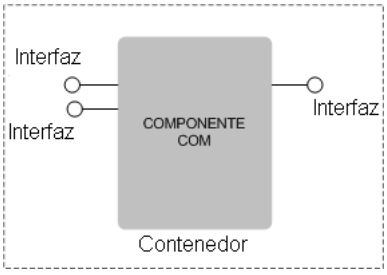

Figura 3.2: Un componente COM

La figura 3.3 ilustra un ejemplo de una interfaz implementada para el componente que permite acceder a los métodos `subirVolumen` y `bajarVolumen`.

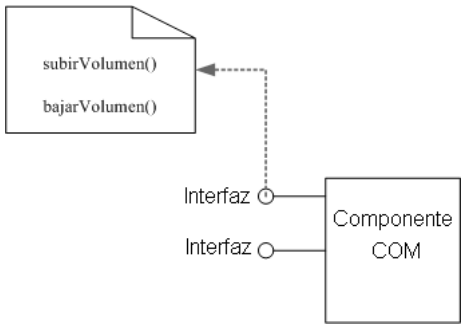

Figura 3.3: Métodos de una interfaz COM

Para la invocación de cualquier método dentro de la interfaz de un componente, cada cliente debe conseguir un apuntador (puntero) a dicha interfaz. Cada elemento COM generalmente presta sus servicios por medio de una o varias interfaces quienes contienen métodos que pueden ser invocados en cualquier momento.

Supongamos que en el ejemplo que hemos venido manejando, un componente cliente (solo por ejemplificar le llamaremos así), necesita un puntero a cada interfaz que requiere, tal y como se muestra en la figura 3.4.

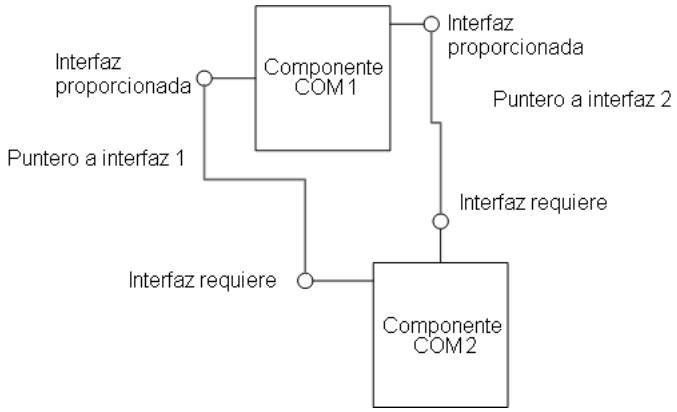

Figura 3.4: Acceso a funciones por medio de la interfaz

Los componentes pueden tener múltiples instancias, ésto depende de la forma en que cada entidad sea utilizada, veamos un ejemplo: piense en un elemento que contiene los métodos de una calculadora básica, es decir, contiene las operaciones de `suma`,

`resta`, `multiplicación` y `división`; mientras otro componente puede utilizar elementos que contienen métodos que permiten realizar el balance general de cualquier empresa.

Ambos componentes deben interactuar entre sí, esto se realiza mediante la interfaz de cada uno, sin embargo, tiene que existir la intervención del sistema operativo, pues dentro del registro del sistema se encuentra una instancia de cada componente. Dichas interfaces pueden ser utilizada por una o varias aplicaciones. En otras palabras se crea un puntero con el cual se puede interactuar para acceder a los métodos que nos interesan.

3.1.3 COM distribuido

La tecnología COM se puede utilizar dentro de una sola computadora o en una red de computadoras. Por extención de COM en ambiente distribuido, DCOM permite compartir componentes COM dentro de una red.

El protocolo COM distribuido (DCOM), extiende las capacidades de COM al admitir la comunicación entre aplicaciones de clientes y componentes de COM a través de los límites entre computadoras, y permite que el desarrollo y la utilización de aplicaciones distribuidas entre computadoras sea más sencillo [40].

Existen dos tipos de servidores COM, según el espacio de direcciones de memoria en donde se encuentren ejecutando. Los servidores COM que se ejecutan en el espacio de direcciones asignado a un proceso cliente se denomina servidor interno al proceso, mientras que los servidores COM que se ejecutan en su propio espacio de direcciones de proceso son llamados servidores externos al proceso.

Físicamente, los servidores internos al proceso se compilan como bibliotecas de vinculación dinámica o *dynamic linking libraries* (archivos DLL, a veces llamados DLLs de ActiveX), mientras que los servidores externos a procesos son compilados como módulos ejecutables (archivos EXE o tambien llamados como EXEs de ActiveX).

3.1.4 Arquitectura DCOM

La manera en que DCOM realiza la comunicación entre procesos y computadoras, es mediante un mecanismo que es implementado por medio de un centralizador de llamadas (proxy) y una replica auxiliar de éste (stub), mediante el uso de la biblioteca COM [26]. La figura 3.5, muestra esta arquitectura.

Cuando una aplicación (el cliente) efectúa una llamada a un componente COM (el servidor), la biblioteca COM (OLE32.dll) lee la información de configuración presente en el registro de configuración del sistema y crea un objeto Proxy en el espacio

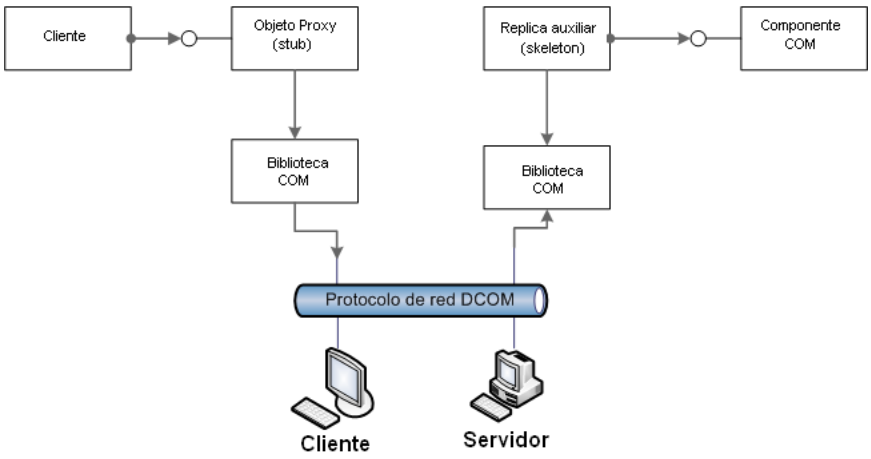

Figura 3.5: Arquitectura DCOM

de direcciones de la aplicaciones cliente. También crea un objeto llamado `réplica auxiliar` en el espacio de direcciones del servidor.

La biblioteca COM permite la comunicación entre estos procesos (o computadoras) empleando ya sea protocolos LPC (*Local Procedure Call*) o RPC (*Remote Procedure Call*), según sea que el cliente y el servidor residan o no en la misma computadora.

El objeto Proxy toma la llamada del cliente, agrupa (empaca) los datos y los envía a través del límite entre procesos o computadoras hacia la réplica auxiliar. El objeto `réplica auxiliar` desagrupa (desempaca) los datos, inicializa el servidor en representación del cliente y efectúa la llamada efectiva para que el servidor la procese.

La biblioteca COM lleva a cabo todas estas tareas de manera transparente tanto para el cliente como para el servidor. La independencia de la ubicación ofrece varios beneficios significativos para las aplicaciones distribuidas entre máquinas, tales como utilización y escalabilidad más sencillas de implementar (escalabilidad debe entenderse como la capacidad que tiene un sistema o aplicación de seguir funcionando bien a medida que va creciendo).

3.1.5 Servicios de DCOM

Servicio a bases de datos

Un sistema de gestión de bases de datos proporciona una forma de organizar, almacenar y obtener información [20]. Un cliente puede acceder a varios DBMS (*Database Management System*) utilizando la misma interfaz, la cual puede ser usada de una forma más general para acceder a diversos datos aunque estos no se encuentren presentes dentro de un DBMS.

La tecnología basada en COM para acceso a bases de datos soluciona el problema de la generalización definiendo estándares de objetos e interfaces para acceder a datos. Esta tecnología proporciona un medio común para que los clientes puedan acceder a los datos almacenados de diversas formas [27].

En realidad OLE Database lo ve todo como objetos COM, de tal forma que una fuente de datos puede modelar un objeto `DataSource` que a su vez tiene un objeto `Command` definido. Este objeto `Command` puede especificar una consulta SQL u otro tipo de comando que permita manejar los datos. Cada objeto `Command` tiene una interfaz que contiene un método `Execute` que ejecuta el comando.

El resultado de cada comando ejecutado es otro objeto llamado `Rowset` que contiene el resultado de la ejecución del comando. Este objeto está provisto de una interfaz con métodos que permiten examinar los datos contenidos. Todos estos objetos se definen usando COM y ofrecen sus servicios a través de interfaces. El resultado es una visión abstracta del acceso a los datos que puede ser implementada de muchas formas y para una gran cantidad de mecanismos de acceso.

La manera en que COM interactúa con los DBMS, es mediando componentes ADO (*ActiveX Data Object*). Vía el uso de estos componentes, es posible tener acceso a cualquier base de datos. Este tipo de componentes permite gestionar el uso de conexiones a bases de datos, así como la ejecución de procedimientos almacenados, el uso del lenguaje SQL desde un componente que permite realizar consultas, inserciones o actualizaciones de los datos.

COM permite desarrollar aplicaciones para acceder y manipular las bases de datos mediante un API (*Application Program Interface*) llamado OLE DB (*Object Linking and Embedding DataBase*).

OLE DB es una especificación que se ha programado en un API de C++ para acceso a datos. Dicha API consiste en un servicio de consumidores y proveedores de datos, donde los primeros toman los datos desde una interfaz OLE DB y los segundos ofrecen datos a una interfaz OLE DB. Este proceso se suele hacer muy rápido debido a que existe una conexión directa a la fuente de los datos.

Para acceder a una base de datos determinada debemos contar con el proveedor OLE DB correspondiente. De esta forma el acceso a una base de datos determinada viene condicionado por la existencia del proveedor OLE DB correspondiente, ver figura 3.6.

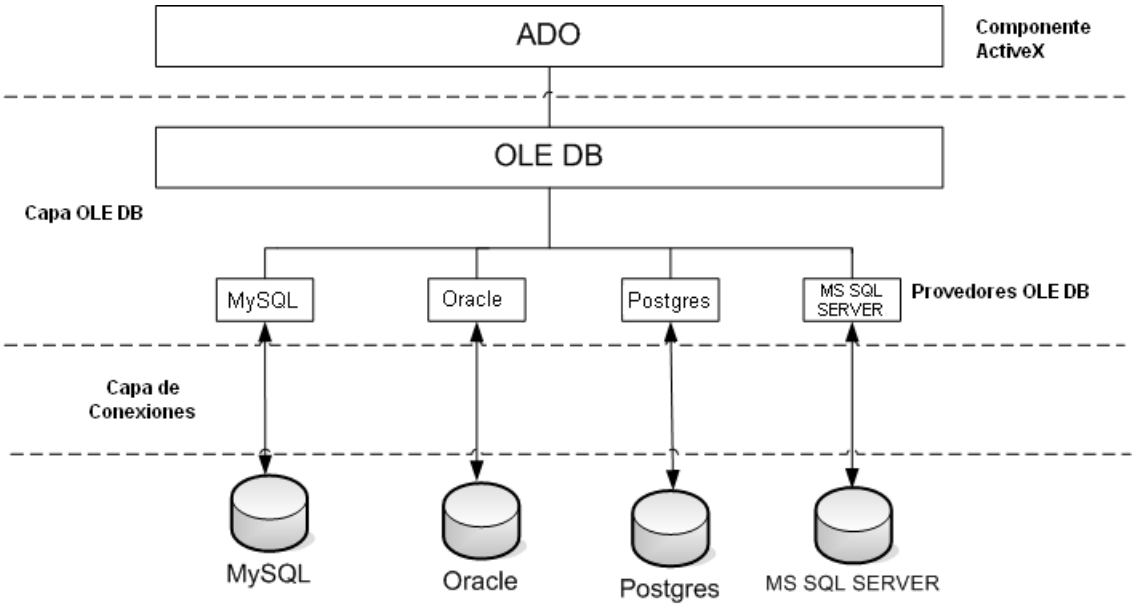

Figura 3.6: Conexión mediante ADO

Servicio de Transacciones

Así como la tecnología de bases de datos de COM lo modela todo usando solo objetos COM el modelo de transacciones basado en COM ve las transacciones como objetos COM. Los objetos definidos incluyen gestores de recursos por ejemplo DBMS, coordinadores de transacciones y transacciones.

Microsoft Transaction Server (MTS)

El MTS es un monitor transacciones, éste es un conjunto de uno o más componentes que brindan el soporte para el diseño, desarrollo, configuración y operación de aplicaciones de transacciones distribuidas [29]. Esto quiere decir que al mismo tiempo es un servidor de componentes, el cual permite definir un modelo de programación por componentes y al mismo tiempo proporciona un ambiente de ejecución.

El MTS soporta componentes que han sido desarrollados en diferentes lenguajes de programación como son: Visual Basic, Visual C++, Delphi, Visual J++, Visual FoxPro e incluso COBOL, en donde los mecanismos de comunicación que son necesarios

para la interacción entre componentes son ofrecidos por el MTS. Este último forma parte del Windows DNA (*Windows Distributed interNet application Arquitectura*) mismo que es utilizado ampliamente en el desarrollo de aplicaciones de *N-capas o N-niveles*.

Elementos del MTS:

- **Object Request Broker** *(ORB)*: se encarga de gestionar todos aquellos componentes que son instaciados por los clientes, es decir, sabe en dónde se encuentran los componentes y cómo es que deben ser creados. Gestiona los procesos de comunicación entre componentes.
- **Monitor transaccional**: comúnmente llamado DTC (*Distributed Transaction Coordinator*), quien se encarga de gestionar el uso de protocolos transaccionales para coordinar las diversas tareas entre componentes, así como el compartir los recursos del sistema entre los diferentes clientes existentes.

Servicio de administración de estados de componentes

Todos los componentes del sistema tienen un estado en determinado momentos, el cual puede ser almacenado para su uso posterior, es decir, el componente puede contener información que puede ser de utilidad en un futuro y por ello es necesario guardar su estado.

Administrador de propiedades compartidas

El SPM (*Shared Property Manager*) permite almacenar propiedades y compartir información entre componentes de un mismo contexto, es decir de un mismo paquete MTS. La información que es almacenada por el SPM no puede ser compartida por aplicaciones MTS distintas.

El SPM está implementado como un proveedor de recursos que almacenan los datos transitorios compartidos en la memoria. También provee grupos de propiedades compartidas que establecen espacios de nombramiento (*namespaces*) relacionados con las propiedades compartidas que contienen, con el fín de evitar colisiones de nombres. El SPM implementa un mecanismo cerrado de bloqueo y serialización para respaldar actualizaciones múltiples y concurrentes, para evitar la pérdida de datos.

Servicios de Directorios

Como las guías o las páginas amarillas de teléfono, un servicio de directorios es un entorno distribuido que permite a sus usuarios buscar información. Se puede buscar tanto información referente a una máquina como referente a una persona. El concepto y la idea es la misma, se puede buscar el nombre de una máquina y podemos

obtener información sobre su dirección en la red o de su procesador; o por el contrario podemos darle el nombre de una persona y nos devolverá su correo electrónico o su número de teléfono.

Debido a que no hay un solo directorio que guarde toda la información que un usuario pueda necesitar existen diferentes servicios de directorios y diferentes tecnologías. Por ejemplo Windows NT usa X.500 el más estándar pero el menos utilizado, Novell definió su propio servicio llamado NDS (Novell Directory Services) y para Internet se ha creado un protocolo de servicio de directorios llamado LDAP (*Lightweight Directory Access Protocol*).

Siguiendo con la filosofía de los dos servicios anteriores OLE *Directory Services* u OLE DS proporciona una interfaz común para acceder a todo tipo de servicio de directorios independiente de su implementación y siguiendo la misma línea se crea una abstracción a partir de objetos COM. De esta forma se puede crear un cliente que pueda funcionar con todos los servicios de directorios.

Servicios Web

La mayoría de las tecnologías creadas por Microsoft en Internet se han basado en COM. Por ejemplo, el navegador de Microsoft, Internet Explorer, está basado en COM y en una extensión de OLE llamada documentos ActiveX, de esta forma un navegador puede visualizar varios tipos de información además de las páginas HTML. Por otro lado la tecnología de controles ActiveX ha sido mejorada para permitir que se pueda descargar controles dentro de un navegador.

Los *scripts* ActiveX proporcionan una forma genérica de ejecutar *scripts* mientras que el estilo de hiperenlaces del Web, basando en *monikers*, permite la creación de hiperenlaces pero no solo entre páginas HTML sino entra cualquier tipo de documentos.

3.2 Paradigma de componentes en DCOM

La especificación COM define un conjunto de parámetros que son necesarios para definir componentes e interfaces de componentes. La programación de un componente en COM se realiza a través de ActiveX.

3.2.1 Interfaces

En capítulos anteriores hemos dicho que una interfaz es un contrato entre un componente y sus clientes (propiamente dicho, es quien hace uso del componente). El componente debe mantener los métodos y atributos tal cual han sido definidos en

la interfaz, esto es, para evitar errores en el momento en que el cliente invoque los métodos del componente aun cuando este haya sido modificado.

El contrato funciona de la siguiente manera: el componente y el cliente se coordinan en la identificación plena de cada una de las interfaces, establecen descriptores comunes de los métodos de cada una de las interfaces así como la manera en que se han de implementar las interfaces.

3.2.2 Identificadores de interfaces

Todas las interfaces que son definidas en COM tienen dos nombres, en donde uno de ellos es definido por y para el uso del programador y el segundo es definido para el uso del *middleware*. Este segundo nombre es único, por lo que se vuelve un identificador que no puede ser repetido en otro componente.

Siempre que un programador desea crear una nueva interfaz ésta va precedida por la letra **I**, así por ejemplo si se quisiera definir la interfaz `suma`, ésta debe ser nombrada como `Isuma` .

Cuando se crea un nombre de programa, éste es único y es comúnmente llamado identificador único global o GUID (*Globally Unique Identifiers*). Un GUID de una interfaz se llama IID (Interface Identifiers).

3.2.3 Descriptores de interfaces

COM realiza la descripción de las interfaces a nivel binario, mediante el uso de IDL (*Interfaz Description Language*), mismo que sigue la especificación del OSF DCE (*Open Software Foundation Distributed Computing Enviroment*).

3.2.4 Implementación de interfaces

La tecnología COM especifica un estándar binario para interfaces, mismo que cada uno de los componentes debe de cumplir. Es de esta forma como se rompe la barrera del lenguaje de programación, pues cada componente puede ser implementado sin problemas [34], lo importante es respetar este estándar binario.

3.2.5 Interfaz IUnknown

Todos los componentes en COM, soportan una interfaz, misma que es heredada de la interfaz `IUnknown`. Esta interfaz contiene los siguientes métodos: `QueryInterface`, `AddRef` y `Release`, mismos que pueden ser accedidos por cualquier interfaz debido a

la herencia existente.

3.2.6 Uso de QueryInterface

Cuando un cliente crea la instancia de un componente, éste obtiene un puntero que es soportado por la `QueryInterface`, con esto lo que realmente se está haciendo es obtener un puntero a una interfaz determinada. Ver figura 3.7.

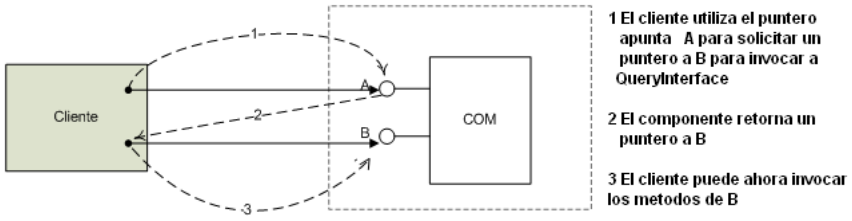

Figura 3.7: QueryInterface

Una de las reglas fundamentales del uso de COM es que la funcionalidad de los componentes no debe cambiar. Si se desea añadir un método a determinado componente, será necesario hacerlo a través de una nueva interfaz para el componente.

3.2.7 Uso de AddRef y de Release

El control del ciclo de vida de un componente se realiza mediante un contador de referencias, esto quiere decir que cada que se crea una referencia a un componente el contador debe de incrementarse. Esto último se realiza mediante el uso del método `addRef`, una vez que el contador de referencias llega a cero, éste tiene que borrar el componente que ha sido utilizado mediante el método `Release`.

3.2.8 Servidor COM

Los componentes son implementados dentro de un servidor. El servidor contiene el código que implementan los métodos de las interfaces y sigue la pista de los datos del componente mientras éste se encuentre activo. Existen tres tipos de servidores de componentes (figura 3.8):

1. *Servidor In-process*: este servidor, también llamado servidor interno al proceso se implementa dentro de una DLL y comparte el mismo espacio de memoria que el cliente.

2. *Servidor Local*: este servidor se implementa en un proceso separado del cliente, aunque ambos se encuentran en la misma computadora.

3. *Servidor Remoto*: este servidor, también llamado servidor externo al proceso se implementa en una DLL o en un proceso separado del cliente. La característica fundamental es que la implementación de éste se encuentra en computadoras diferentes.

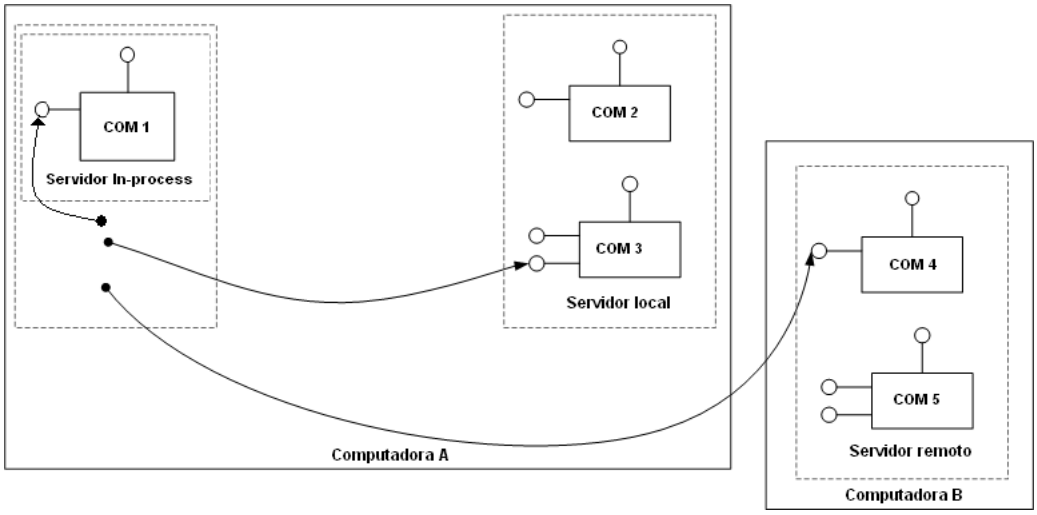

Figura 3.8: Servidores COM

Los clientes acceden a nivel de programación de forma transparente a los componentes. Esta característica se debe a que los clientes siempre manejan punteros que hacen referencia al componente que utilizan, los mecanismos de comunicación que se utilizan son gestionados por el *middleware*.

3.2.9 Biblioteca COM

Búsqueda de servidores: cuando un cliente realiza una petición a la biblioteca COM para implementar un componente, éste le pasa el identificador de la interfaz a la biblioteca para localizar el servidor del componente. Para realizar esta actividad se basa en el registro del sistema, en donde se encuentran registrados los componentes existentes.

3.2.10 Proceso de implementación de un componente

Para implementar un componente se llama a la función `CoCreateInstance` de la biblioteca COM. Entonces la función localiza el componente solicitado dentro del

registro del sistema, revisa su localización y el tipo de servidor que debe ejecutar. A su vez el servidor crea una instancia del componente y devuelve un puntero a la interfaz requerida.

La biblioteca COM devuelve este puntero como valor de retorno de la función `CoCreateInstance`. En la figura 3.9 se puede observar el proceso que hemos descrito. Es importante hacer énfasis que en la utilización de componentes no existen constructores ni destructores de componentes ni tampoco existen mecanismos de inicialización de componentes.

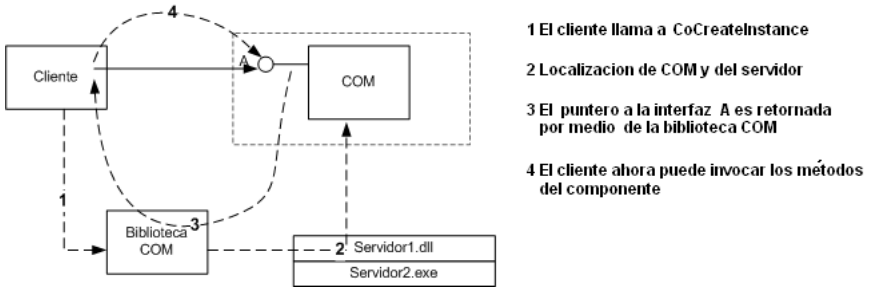

Figura 3.9: Creación del componente COM

3.2.11 Reutilización de componentes

El principal objetivo del paradigma orientado a componentes es la reutilización, por ello es que COM crea dos mecanismos para implementar el reuso: delegación y agregación. Para ello se definen nuevos componentes, llamados "externos", mismos que reutilizan los servicios de otros componentes llamados "internos".

- *Delegación:* en este mecanismo, los componentes externos contienen métodos que invocan a los componentes internos de forma directa, ver figura (3.10).

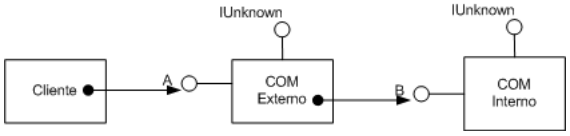

Figura 3.10: Delegación en COM

- *Agregación:* este mecanismo expone directamente las interfaces del componente interno en el componente externo, en donde el primero llama la interfaz `IUnknown` del segundo con lo cual se permite la comunicación entre un cliente y los métodos de un componente interno. Vease la figura (3.11)

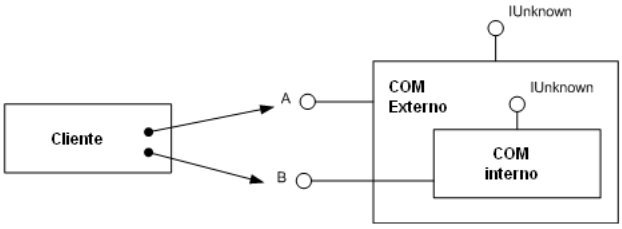

Figura 3.11: Agregación en COM

3.2.12 Componentes remotos

Independientemente de donde se puede ejecutar un componente, un cliente generalmente lo crea y después adquiere un puntero a su interfaz debido a que necesita de este último para poder usar el componente (figura 3.9). La mayoría de los componentes son implementados mediante DLL, es decir que primero se hace un llamado a `CoCreateInstance` y luego se usa el `QueryInterface` para pedir un puntero a la interfaz que nos interesa. Cualquier cliente puede crear un componente en un servidor remoto usando la misma llamada, el cliente incluso no necesita preocuparse de que el componente se esta ejecutando en otra computadora.

Cuando se utilizan componentes remotos, se debe especificar en que máquina debe ser creado el componente, es decir que debe ser registrado el nombre de la computadora en donde se encuentra físicamente el componente, ya sea un DLL o un EXE, según corresponda el caso.

Para crear el componente remoto, se contacta con la máquina que ha sido registrada y se busca en su registro de sistema el componente que nos interesa. El componente es ejecutado de forma remota respondiendo a las peticiones que se le hayan hecho.

DCOM permite varias formas de identificar una computadora, dependiendo del protocolo de red que se utilice. DCOM permite utilizar los nombre de dominio de TCP/IP y las direcciones IP, nombre NetBIOS y nombres usados en las redes Novell. Cuando la máquina remota se ha contactado, el objeto se crea allí usando la información del componente que ha sido encontrada en el registro.

3.3 CORBA

En la presente sección daremos una pequeña introducción a los conceptos fundamentales de la arquitectura CORBA (*Common Object Request Broker Architecture*). Esta tecnología es una de las más robustas que existen para la implementación de sistemas distribuidos [11]. La especificación de CORBA fue publicada por la OMG (*Object*

Management Group) [47] y está basada en el modelo de objetos descrito por la OMA (*Object Management Architecture*). Trataremos de explicar a grandes rasgos esta arquitectura.

3.3.1 Arquitectura de manejo de objetos

La *Object Management Group*, fue fundada en 1989 como un consorcio internacional no lucrativo y está compuesto por miembros del campo de la tecnología informática alrededor del mundo. La principal actividad de la OMG consiste en la publicación y actualización de las especifícaciones que describen a las infraestructuras orientadas a objetos [4]. Las publicaciones de la OMG son libres, éstas especifican y coordinan las contribuciones y modificaciones de los estándares realizados por los miembros del consorcio.

Una de las especifícaciones importantes de la OMG, es la OMA [17] que describe la plataforma general para el desarrollo de aplicaciones distribuidas orientadas a objetos. Por otro lado existe CORBA [17] que es una especialización de OMA.

La principal característica de OMA resulta ser que es un modelo de *objetos abstracto* y una *arquitectura de referencia*. El modelo de objetos de OMA diferencia entre *objetos semánticos* y *objetos de implementación*. Los objetos semánticos describen las características de los objetos que están en el exterior y que son visibles para los clientes, los objetos de implementación se ocupan de los conceptos necesarios para la ejecución de los objetos.

La OMA pone principal énfasis en los objetos semánticos, y el aspecto de los objetos de implementación es sólo definido para permitir una amplia flexibilidad en la implementación de objetos.

Mientras que el modelo de objetos de OMA provee una identificación abstracta de objetos, la arquitectura de referencia define relaciones entre objetos. En la figura 3.12 se muestra la estructura de la OMA. El componente principal de ésta es el ORB (*Object Request Broker*) o intermediario entre las peticiones y los objetos.

Éste funciona como un bus de software, es decir permite la comunicación entre objetos en cuatro diferentes categorías:

- **Servicios de objetos**: esta categoría combina los servicios horizontales del sistema que son independientes de la aplicación y pueden ser utilizados en diferentes contextos. Estos servicios son fundamentales para distribuir aplicaciones CORBA. Ejemplos de ésto pueden ser los servicios de nombres, de transacciones y de seguridad.

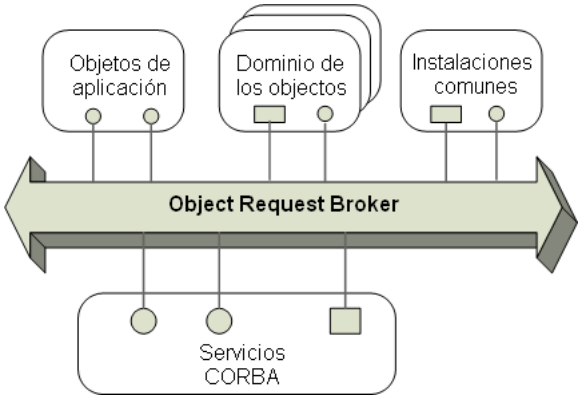

Figura 3.12: Arquitectura OMA

- **Instalaciones comunes**: las instalaciones comunes proveen servicios horizontales a los usuarios finales, que son típicamente requeridos en diferentes contextos de la aplicación, estos servicios no son fundamentales como los servicios de objetos, comunmente son una colección de servicios que las aplicaciones pueden compartir. Un ejemplo de éste es el servicio de impresión.

- **Dominio de objetos**: el dominio de objetos representa servicios verticales para áreas específicas de la aplicación. Un ejemplo del dominio de objetos son los servicios de telecomunicaciones y financieros.

- **Objetos de aplicaciones**: los objetos de aplicación representan servicios específicos para la aplicación. En contraste con las otras tres categorías, éstos no han tenido un fuerte impacto dentro de la OMG.

3.3.2 Common Object Request Broker Architecture

CORBA es derivado como una instancia de la OMA presentada en la sección anterior. La figura 3.13 representa una visión general del *middleware* CORBA-COMPONETS. Los componentes que son parte de CORBA están sombreados en gris y los componentes de la aplicación son los que se encuentran con fondo blanco.

3.3.3 Modelo de objetos de CORBA

La diferencia entre el modelo de objetos de CORBA y el modelo de objetos de OMA es que el primero transforma el modelo abstracto de objetos del segundo a una forma concreta. Por ejemplo, en contraste a OMA, CORBA define un número básico de tipos con el cual se pueden construir estructuras, módulos, clases o componentes. Por otro lado CORBA define un servicio de designación y de invocación semántica de las operaciones definidas dentro de una interfaz. La designación de una operación

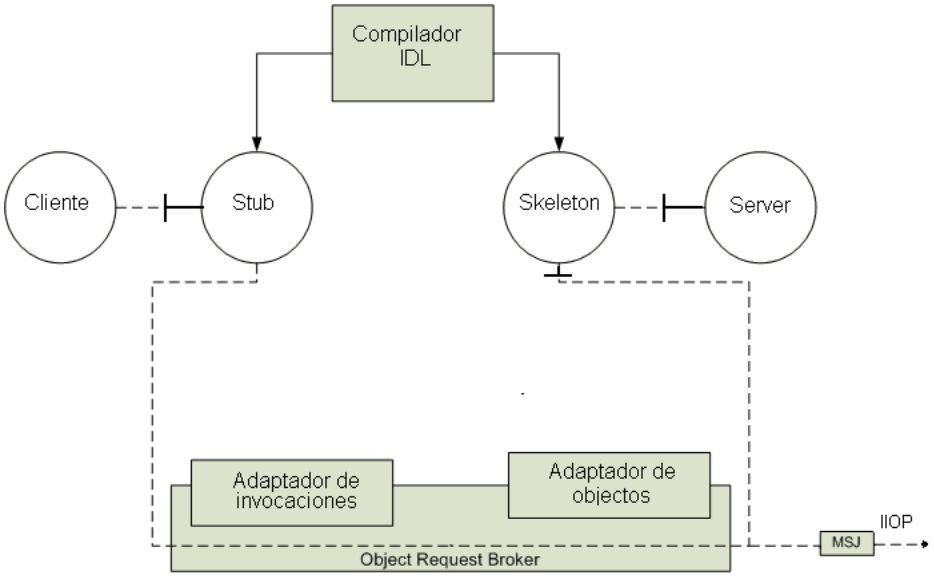

Figura 3.13: Arquitectura de un sistema basado en CORBA

consiste de:

- **Invocación semántica**: especifica la estructura semántica que es usada por alguna operación.
- **Tipo de resultado**: define el tipo de valor de retorno, que también puede ser de tipo `void`. El valor retornado es el mismo parámetro de salida
- **Nombre de la operación**: indica el nombre de una operación. Un nombre consiste de un identificador y debe ser único dentro de la interfaz.
- **Lista de parámetros**: cada parámetro tiene un tipo y una etiqueta que lo identifica, así como si es un parámetro de entrada, salida o una combinación de ambos.
- **Lista de excepciones**: contiene una lista opcional del tipo de excepciones que pueden ocurrir durante alguna operación. Una excepción señala un error en el llamado de la operación.
- **Lista de contexto**: contiene una lista opcional del contexto de información. Éste consiste en un conjunto de parámetros implícitos que son definidos por el cliente antes de cualquier invocación de una operación y transmitida junto con los parámetros de invocación.

Las características anteriores son un esbozo del modelo formal de CORBA, la especificación formal esta dada por la IDL (*Interface Definition Language*), ésta permite modelar las interfaces de objetos que serán utilizados en la aplicación.

3.3.4 Lenguaje de definición de interfaces

El IDL [17] es usado para especificar interfaces de objetos independientemente de un lenguaje de programación. Esto hace que el IDL sea la base que separa la interfaz de la implementación del objeto. El IDL-CORBA es un lenguaje declarativo.

La sintaxis del IDL está basada ampliamente en el lenguaje C++, sin embargo incluye algunas construcciones adicionales para utilizar características especiales de los entornos distribuidos [25], por ejemplo la identificación de parámetros de entrada o de salida.

Los elementos de CORBA-IDL permiten la definición de tipos que están conformados por el modelo de objetos de CORBA. El IDL provee mecanismos de herencia de interfaces para permitir la existencia de tipos que pueden ser reutilizados en la construcción de nuevas interfaces.

La herencia de interfaces solo permite la reutilización de interfaces. Existen mecanismos apropiados para la implementación de interfaces para diferentes lenguajes de programación, que pueden usar código existente de un programa para la construcción de un nuevo objeto CORBA.

3.3.5 Mapeo de lenguaje IDL

El mapeo de interfaces IDL a un lenguaje particular de programación como C++ es definido por un lenguaje de mapeo IDL. La OMG actualmente define lenguaje de mapeo para C, C++, Java, Smalltalk, Python, Ada 95, COBOL y PL/1. Un compilador IDL automatiza el proceso de mapeo, utilizando la especificación correspondiente al lenguaje de programación que se requiera. Cada interfaz IDL es mapeada en dos proxies: un *stub* y un *skeleton*, el primero se encuentra en el cliente y el segundo en el servidor.

Los detalles del lenguaje de mapeo IDL a un determinado lenguaje de programación, dependen de éste último. Por ejemplo, en un lenguaje de programación orientado a objetos, la interfaz IDL será mapeada a clases, en un lenguaje estructurado será a estructuras y funciones.

3.3.6 Object Request Broker

El ORB (*Object Request Broker*) trasmite operaciones por medio de invocación de un cliente a un servidor, que puede ser localizado en: a) el mismo espacio de direcciones, b) en diferentes espacios de direcciones en la misma computadora o c) en diferentes computadoras. El ORB asegura que la comunicación entre objetos y un ambiente distribuido sea transparente.

Existen unos componentes especiales que son utilizados por las interfaces entre el ORB y la aplicación. Por el lado del cliente el adaptador de invocación permite que una operación sea generada e invocada. De manera similar el adaptador de objetos en el servidor permite que una invocación sea entregada en la implementación del objeto. La tarea del ORB es aceptar operaciones del adaptador de invocaciones que transmitirá y entregara de forma apropiada al adaptador de objetos.

La arquitectura de interoperabilidad del ORB, es basada en el GIOP (*General Inter ORB Protocol*) que está representado en la figura 3.14. El GIOP específica la sintaxis de transferencia y un conjunto de formatos estándar para el ORB sobre cualquier conexión orientada a transporte. El IIOP (Internet Inter ORB Protocol) especifica como el GIOP es construido sobre el protocolo TCP/IP para realizar el transporte de mensajes, objetos y componentes.

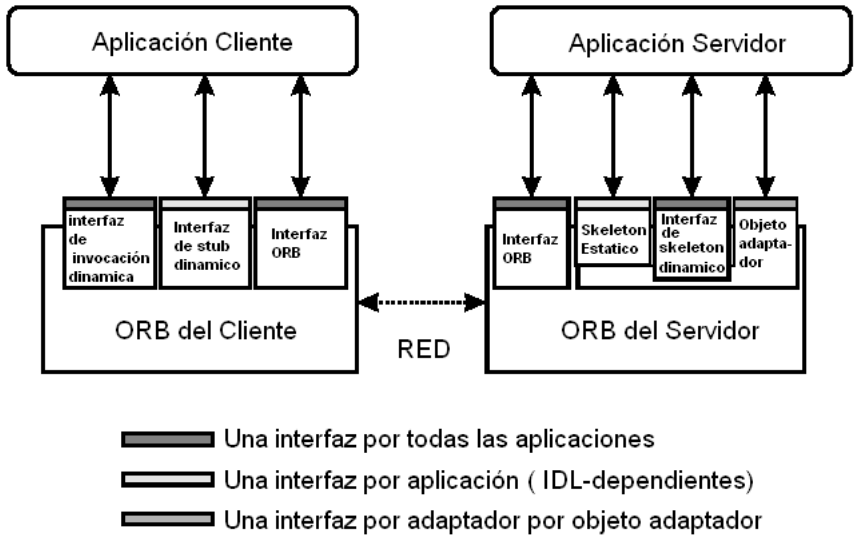

Figura 3.14: Arquitectura del ORB

Todas las llamadas ORB que sean realizadas y que se deseen hacer compatibles entre objetos o componentes CORBA se deben realizar mediante la implementación nativa del IIOP o bien proveer un puente intermedio para lograr la comunicación. La

arquitectura de interoperabilidad del ORB permite que se utilice otro protocolo de interoperabilidad en casos muy especiales, el ESIOP (*Enviroment Specific Inter ORB Protocol*).

3.3.7 Invocador y adaptador de objetos

Los clientes usan el adaptador de invocaciones, ya sea de forma indirecta a través de objetos *stub* o directamente en orden de transferencia del invocador de métodos del ORB. El adaptador de invocaciones es un componente que está separado del ORB porque la funcionalidad requerida para inicializar los objetos por el invocador de métodos puede variar considerablemente.

De forma similar al adaptador de invocaciones, el adaptador de objetos implementa la conexión entre el ORB y el objeto de implementación. Ambos son componentes separados del ORB porque diferentes tipos de objetos de implementación tienen diferentes demandas sobre el adaptador de objetos. Este último maneja el ciclo de vida de los objetos CORBA y maneja la ejecución del operador de invocaciones. Consecuentemente los objetos de implementación son llevados al adaptador de objetos a través de los *skeletons*.

3.3.8 Servicios comunes de CORBA

El **servicio de nombres** de objetos de CORBA, permite a los objetos que están sobre el bus localizar a otros objetos mediante un nombre. También soporta federación de contextos, es decir, agrupación de diferentes servidores para ofrecer un servicio común, de tal forma que si el nombre no se encuentra en un ORB debe ir a buscar a otro. Este servicio permite a los objetos, ser enlazados con los directorios actuales de red o con los contextos de nombres actuales como ISO X500, OSF Dce, Sun NIS+ y el LDAP para Internet.

El **servicio de gestión** de objetos proporciona un servicio de páginas amarillas para los objetos CORBA, de tal forma que se facilitara la búsqueda y localización de objetos determinados en función de palabras claves, contenidas dentro de la descripción intrínsecamente de un contexto determinado.

El **servicio de transacción** de objetos es posiblemente el servicio más importante, ya que, es el que nos garantiza la construcción de aplicaciones distribuidas de forma robusta. Permite transacciones simples y opcionalmente anidadas, es un coordinador de dos fases. Las operaciones de transacción son `begin`, `commit` y `rollback` que se definen en el contexto actual de tal forma que, un objeto podrá realizar transacciones simplemente heredando de la clase `TransactionalObject`.

Con las transacciones podremos, desechar el conjunto de operaciones invocadas después de `begin` si ocurre algún fallo mediante `rollback`, sin afectar a la integridad del sistema, o por el contrario, si todo sale correctamente podremos acabar la transacción con `commit`.

El **servicio de control de concurrencia** provee las interfaces para bloquear y liberar bloques que permitan a múltiples clientes coordinar su acceso a recursos compartidos. Estos bloqueos pueden ser tanto en el contexto de una transacción como en el contexto de recursos independientes. En el primer caso se produce una liberación cuando se ejecuta el método `rollback`. Se puede definir un `lockset` que es una colección de bloqueos asociados a un recurso simple. A su vez se define un coordinador de bloqueos que gestiona la liberación de los `locksets` relacionados.

El **servicio de persistencia** de objetos de CORBA proporciona una interfaz sencilla para almacenar objetos persistentemente en varios servicios de almacenamiento, desde bases de datos orientadas a objetos (ODBMS), hasta bases de datos relacionales (RDBMS), pasando por archivos simples, o sistemas estructurados de almacenamiento, usados en los documentos compuestos.

El POS (*Persistence Object Service*) puede soportar almacenamientos de un nivel y de dos niveles. En el primero, el cliente no está al tanto si el objeto está en memoria o en disco. En contraste, en un almacenamiento de dos niveles separa la memoria del almacenamiento persistente. El objeto debe ser explícitamente colocado de una base de datos a memoria y viceversa.

El **servicio de consultas** ayuda a buscar objetos a través de sus atributos. Es similar al servicio de gestion de objetos, pero en lugar de localizar servidores, localiza instancias. Las consultas están basadas en los atributos que un objeto hace públicos o accesible a través de sus operaciones. Se han definido dos lenguajes de consulta: el *Object Database Management Group* (ODMG-93) define el OQL y el SQL con extensiones de objetos.

El **servicio de relación** proporciona una forma de crear asociaciones dinámicamente entre objetos que a priori no tiene relación uno con el otro, así como mecanismos para recorrer estos enlaces. Se puede usar este mecanismo para garantizar las restricciones de integridad referencial, mantener un seguimiento de la relación de contención o para cualquier tipo de enlaces entre componentes.

El **servicio de propiedades** de objetos permite asociar pares de la forma "nombre-valor" a cualquier objeto. El servicio de propiedades no interpreta el valor de las propiedades, son útiles cuando se necesita pegar una información a un objeto arbitrario cualquiera, e.g. identificar su estado, ponerle un título a un objeto, añadirle fecha de creación, etc.

El **servicio de tiempo** del objeto proporciona interfaces para sincronizar el tiempo en un sistema distribuido, así como las operaciones necesarias para gestionar eventos disparados por tiempo.

El **servicio de seguridad** provee de un marco completo de seguridad para objetos distribuidos permitiendo: autentificación, control de acceso, confidencialidad y auditorias.

El **servicio de gestión de cambio de versión** de objetos permite una evolución de los sistemas de forma segura, es decir poder dar seguimiento a la evolución de las diferentes versiones que un objeto va tomando, pero que a su vez los objetos que usaban versiones antiguas de ese objeto no se vean afectados.

El **servicio Web** no son soportados por CORBA, sin embargo, dentro de la especificación existen una serie de protocolos que son equiparables. Por poner un ejemplo, tanto en los Servicios Web como en CORBA, existe la descripción de interfaces, para el primer caso se utiliza el WSDL (*Web Service Definition Language*) y para el segundo caso existe el IDL (*Interface Description Language*). En este estudio no hacemos una profundización del tema, debido a que no es nuestro objeto de estudio.

3.4 Paradigma de componentes en CORBA

CORBA 3.0 introdujo el Modelo de Componentes de CORBA o *CORBA Components Model* (CCM), que permite desarrollar e implementar componentes y ensamblar sistemas a base de componentes CORBA.

3.4.1 Modelo de Componentes CORBA

Un componente CORBA [18] es una extensión de un objeto CORBA, basado en un modelo de programa servidor diferente. CCM es el marco sobre el cual se definen, implementan y desarrollan componentes en CORBA. Este marco aborda todas las etapas de desarrollo de la aplicación, es decir desde el diseño hasta la implementación de la aplicación.

El principal propósito de CCM es reducir el esfuerzo necesario para desarrollar e implementar servidores CORBA. Esta simplificación puede observarse desde las siguientes perspectivas [32]:

- **Abstracción funcional del POA:** funcionalmente el POA (*Portable Object Adapter*), es implementado mediante un contenedor de componentes. El contenedor es responsable de administrar el ciclo de vida del componente.

- **Abastecimiento de servicios CORBA:** los servicios CORBA, típicamente requeridos por aplicaciones empresariales, incluyen administración de seguridad, manejo de eventos y transacciones. El CCM especifica que estos servicios deben estar disponibles para los componentes y definir interfaces simples que cubran las características más comunes por los servicios.

- **Soporte para las instancias de componentes:** el CCM soporta el desarrollo de componentes reutilizables de propósito general. De esta forma una clara separación puede hacerse entre la lógica del componente y la funcionalidad de CORBA.

El modelo de componentes de CORBA soporta la configuración y el ensamble rápido y fácil de nuevas aplicaciones mediante la integración de componentes reutilizables [6].

Un componente en CORBA es una entidad de software que puede ser implementada y ejecutada por sí misma, ésta consta de las siguientes características:

1. Puede ser compuesta por cero, uno o más componentes.

2. Tiene interfaces

3. Tiene atributos

4. Tiene puertos.

Existen dos clases de puertos, *puertos de interfaz* y *puertos de eventos*. Los puertos de interfaz pueden ser *provistos o requeridos*, y los puertos de eventos son *eventos generados* y *eventos recibidos*. En la figura 3.15, se observa la representación de un componente en CORBA.

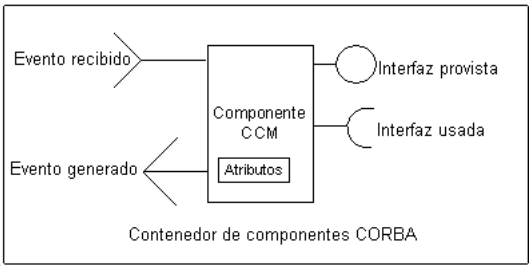

Figura 3.15: Representación de un componente CORBA

Con respecto al lenguaje IDL para el modelado de componentes, se han introducido nuevas palabras clave para cada tipo de puerto que se esté utilizando:

- **Provisto:** un puerto de interfaz provista, solo puede llamar a una interfaz específica que provea algún resultado o dato. El componente implementa esta interfaz, la cual es declarada como una interfaz regular en lenguaje IDL, que contiene operaciones y atributos.

- **Usado:** un puerto de interfaz requerida, solo puede llamar una sola interfaz específica que sea capaz de recibir un resultado o dato. Esto implica una dependencia sobre otro componente. En un ensamble de componentes, el componente debe ser conectado con otro componente a través de la interfaz provista, solo así el componente puede acceder a la funcionalidad del otro componente.

- **Publicado:** son puertos de eventos generados en *broadcast*.

- **Emitido:** son puertos de eventos generados en *unicast*.

- **Consumido:** son puertos de eventos recibidos.

Mientras los componentes en el modelo abstracto no tienen operaciones propias, debido a que su funcionamiento solo es publicado por medio de sus interfaces, el modelo de componentes de CORBA sólo permite componentes que soporten una o más interfaces.

Esta característica, existe en gran medida, para un uso extenso de la herencia entre componentes, en donde un servicio existente se pone a disposición como un componente.

3.4.2 Arquitectura básica del CCM

La arquitectura básica (ver figura 3.16) consiste de los siguientes elementos:

- **Componentes:** es el elemento básico para aplicaciones de servidor

- **Componentes *Home*:** es una factoría de objetos que crea y administra componentes

- **Contenedor:** es un entorno de soporte de componentes, el cual permite su activación y ejecución.

3.4.3 Componentes

Hay que notar que la especificación de CORBA hace una clara distinción entre dos tipos de componentes básicos y extendidos:

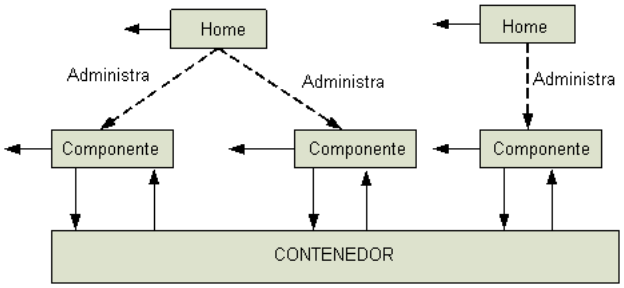

Figura 3.16: Arquitectura básica del CCM

Componente básico: conforma el nivel básico del CCM. Es conceptualmente similar a los objetos CORBA, cada instancia de un componente soporta una o más interfaces IDL. Los desarrolladores de aplicaciones implementan los componentes únicamente con operaciones y atributos. Esta es la principal diferencia entre un componente y un objeto CORBA.

Componente extendido: conforma el nivel extendido del CCM. Este tipo de componentes tiene características adicionales, tales como son los llamados *puertos* que son una forma de interactuar enre los clientes, algunos elementos del ambiente de la aplicación y los componentes. Los componentes extendidos no son soportados ni por los componentes básicos ni por los objetos CORBA.

Una característica notable de los componentes extendidos es que éstos pueden soportar múltiples y distintas interfaces.

3.4.4 Componentes *Home*

Un componente *home* es un objeto que administra un tipo particular de componente. Cada tipo de componente es administrado por lo menos por un componente de tipo *home*. Inversamente cada componente *home*, administra uno y solamente un tipo de componente.

Un componente *home*, típicamente permite realizar las siguientes acciones en cuanto a la administración de componentes:

- crear instancias,
- encontrar instancias, y
- remover instancias.

Los componentes *home*, son análogos a una fábrica de objetos. La idea principal para introducir este tipo de componentes se basa en tener la certeza de administrar de manera correcta el ciclo de vida de los componentes dentro del servidor.

3.4.5 Contenedor

Un contenedor es el ambiente en donde los componentes están embebidos, incluyendo interfaces y servicios estándar, como son transacciones y seguridad. El contendor se encuentra en una capa arriba del POA y ofrece políticas para la comprobación de la administración del ciclo de vida de los componentes.

Existen dos formas de interacción entre los componentes y el contenedor:

- **El componente invoca unas operaciones sobre el contenedor:** un componente puede usar los servicios suministrados por el contenedor o invocar operaciones sobre el contexto de éste.
- **El contenedor invoca unas operaciones sobre el componente:** todos los componentes heredan de una interfaz base, que define el retorno de cualquier operación que el contenedor haya invocado.

Un número de interfaces IDL son definidas por medio de la interacción entre el contenedor y los componentes embebidos.

3.4.6 Categorías de componentes

Las categorías de componentes en CCM son patrones comunes para implementar componentes. El desarrollo de una aplicación debe escoger una de las siguientes categorías de componentes permitidos:

- Sesión
- Servicio
- Entidad
- Proceso.

3.4.7 Componentes de sesión

Un componente de sesión es típicamente un objeto temporal que hace los trabajos de representante del cliente. Por ejemplo una instancia de un tipo particular de un componente de sesión es frecuentemente creada por cada cliente que se conecta al servidor.

Después de que el cliente se desconecta del servidor, el componente de sesión ya no es necesitado y puede ser descartado. Las características básicas de un componente de sesión son:

- su estado temporal,
- su identificación temporal, y

- el componente *sesión* no puede participar en servicios de transacción OTS (*Object Transaction Services*).

3.4.8 Componentes de servicio

Un componente de servicio, así como los componentes de sesión son objetos temporales que hacen los trabajos de representante del cliente. Sin embargo, un componente de servicio es particularmente simple porque éste no tiene estado.

Se deduce que un solo componente de servicio puede servir a cualquier número de clientes. Las caracteristicas básicas de un componente de sesión son:

- no tiene estado,
- no tiene identidad, y
- no puede participar en servicios de transacción OTS.

3.4.9 Componentes de entidad

Un componente de entidad puede ser usado para representar datos almacenados en una base de datos. Por ejemplo, cada objeto cliente o producto puede ser modelado como un componente de entidad.

El contenedor de componentes tiene que administrar los estados de los componentes de entidad, cargando y almacenándolo cuando sea necesario. Las características básicas de estos componentes son:

- su estado es persistente,
- su identidad es persistente y es automáticamente visible para los clientes, y
- puede participar en servicios de transacción OTS.

3.4.10 Componentes de proceso

La principal diferencia entre un componente de entidad y un componente de proceso es la accesibilidad que brinda el componente de entidad. Un componente de proceso no expone su identidad a los clientes.

Un componente de proceso es utilizado para representar procesos, tales como son la venta o compra de artículos, donde el estado del proceso necesita ser almacenado

de forma persistente o el proceso participa en una transacción distribuida. Existen algunos casos en donde no existen transacciones persistentes o distribuidas pero son requeridas. Las características básicas de estos componentes son:

- su estado persistente,
- su identidad es persistente, pero no es visible para los clientes, y
- puede participar en servicios de transacción OTS.

3.4.11 IDL equivalente

Aunque en tiempo de ejecución, no es visible la implementación de componentes, un ORB es usado para invocar las operaciones sobre las interfaces y la transportación de eventos entre componentes.

Para la interoperabilidad, el ORB es también utilizado durante la configuración para interconectar puertos. Para este propósito, el modelo de componentes de CORBA utiliza un IDL equivalente.

Existen un conjunto de reglas para definir *interfaces equivalentes* para componentes y componentes *home*. Una interfaz equivalente de un componente contiene todos los atributos del componente y "operaciones equivalentes" para cada uno de los puertos.

Por ejemplo, si un componente tiene una interfaz llamada `comida`, esta interfaz contendrá una operación llamada `proveedor_comida` que retorna una referencia del objeto con el tipo de dato apropiado. La interfaz equivalente solo hereda todo lo que soporte la interfaz original y el componente.

En tiempo de ejecución, el software puede usar interfaces equivalentes de componentes para acceder a referencias de objetos para cada interfaz de evento generado y pasar éstos a una interfaz de evento receptor.

La implementación de los componentes es responsabilidad del contenedor de componentes, pues genera las referencias necesarias que pueden ser utilizadas por la aplicación desarrollada o bien por otros clientes, según corresponda el caso.

3.4.12 CIDL y PSDL

Adicional al lenguaje de mapeo para componentes, CCM también define el Lenguaje de Definición e Implementación de Componentes o *Component Implementation Definition Language (CIDL)*.

CIDL es basado sobre el Lenguaje de Definición de Estado Persistente o *Persistent State Definition Language(PSDL).*

CIDL trata dos ideas ortogonales. Es decir, permite la descripción de la implementación de *segmentos* y *composición* de componentes. Esta característica de CIDL permite la generación automática de un ejecutor (la implementación de un componente que puede ser utilizado por el contenedor) que delega operaciones sobre el componente o interfaz deseado.

Heredando de las características del PSDL, CIDL permite la descripción del estado de un componente, para identificar variables miembro que necesitan ser preservadas a través de reiniciar componentes. Un contenedor de procesos o entidades puede utilizar esta información para automatizar el almacenamiento y la recuperación del estado de componentes.

3.5 J2EE

3.5.1 Introducción

La plataforma Java Enterprise Edition [43], ha sido desarrollada por SUN para el desarrollo de aplicaciones empresariales que deseen aplicar una arquitectura basada en la Web. El uso de esta tecnología ofrece algunas ventajas:

- **Soporte multiplataforma**: debido a que el diseño de J2EE ha sido basado en el lenguaje Java, la implementación de los diseños realizados sobre esta arquitectura pueden ser ejecutados e implementados en cualquier plataforma de sistema operativo.
- **Competitividad**: las características de rendimiento y precio ofrecidas por esta arquitectura, permite que exista un gran número de soluciones basadas en esta tecnología para satisfacer los requerimientos del cliente.
- **Madurez**: el rápido crecimiento que esta tecnología ha tenido en los últimos años, ha provocado que tenga un grado de madurez comparable con tecnologías como CORBA o como el Microsoft Transaction Server (MTS).
- **Soluciones libres**: la mayor bondad que puede tener esta tecnología en la actualidad es que permite que todo el desarrollo pueda ser basado exclusivamente en software libre, y en ello va inmersa la reutilización de código, misma que los arquitectos de software utilizan frecuentemente para realizar sus desarrollos.

En algunas ocasiones se puede confundir la arquitectura J2EE con un kit de desarrollo, tal y como sucede con el JDK, sin embargo Java Entrerprise Edition es más que eso, debido a que es un conjunto de arquitecturas muy robustas que permiten desarrollar poderosas aplicaciones Web.

3.5.2 El estándar J2EE

La especificación J2EE define un conjunto de servicios que una aplicación servidor debería soportar, y como es que las API pueden acceder a los servicios prestados [2].

Es importante hacer notar que J2EE no es un solo producto, si no que es una especificación la cual contiene tres elementos diferentes. Estos elementos, son componentes que los desarrolladores pueden utilizar para realizar sus sistemas:

- los Servlet,
- los JSP's y,
- los EJB's.

A continuación daremos una pequeña introducción de cada una de estas tecnologías.

3.5.3 Java Servlets

Los *Servlets* proporcionan métodos que permiten escribir programas Java del lado del servidor. Un uso común de los *servlets* es la generación dinámica de páginas Web. Los *servlets* han reemplazado a los tradicionales CGI (Common Gateway Interface), como métodos para la generación de páginas Web interactivas. El uso generalizado de CGI es muy popular, sin embargo, los CGI's sufren de problemas de escalabilidad, generalmente son escritos usando Script Shell, C o C++.

3.5.4 Java Server Pages (JSP)

Los JSP permiten realizar diseños para Web, construyendo páginas Web interactivas sin tener que profundizar en los detalles del lenguaje Java. Un JSP se parece en gran medida a una página HTML, por lo que cualquier programador de páginas Web en lenguaje HTML no debe sentirse desorientado al tener que trabajar bajo esta tecnología: la gran diferencia entre entre tener JSP y HTML es que el primero permite insertar fragmentos de código Java dentro de la página, sitio o sistema que se ésta desarrollando.

3.5.5 Enterprise Java Beans (EJBs)

Los EJB son quizás los más importantes de los tres tipos de elementos que maneja el J2EE, por ello es que dentro de nuestro estudio nos enfocaremos a estudiarlos, analizarlos e implementarlos.

Un EJB es una clase Java que posee varias características especiales, entre las cuales destacan las siguientes:

- Distribución
- Consistencia de transacciones
- Manejo de multihilos
- Persistencia.

Muchas de las características de un EJB son ofrecidas libremente por el servidor de aplicaciones [3]. En este caso el programador puede despreocuparse de los detalles de la programación de los servicios ofrecidos, debido a que éstos son proporcionados por la misma arquitectura y se presentan de forma transparente para ser utilizados. Por lo tanto el programador deberá únicamente concentrarse en la labor de implementación del diseño.

3.5.6 La arquitectura de J2EE

La arquitectura de J2EE estandariza el desarrollo de aplicaciones empresariales distribuidas. Esta es una plataforma basada en componentes distribuidos. La figura 3.17 muestra la arquitectura de múltiples niveles para el desarrollo de aplicaciones [5].

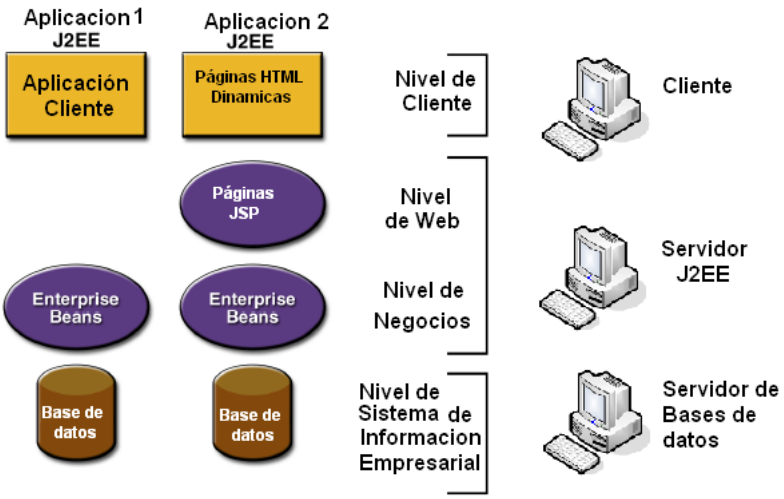

Figura 3.17: Aplicación multinivel

La arquitectura J2EE es definida por la especificación J2EE, que ha sido desarrollada bajo el control de Sun Microsystems. Esta tecnología ha permitido explorar varios aspectos en los ambientes distribuidos de software.

Cada parte de la arquitectura J2EE se diseña bajo un nivel específico, así las aplicaciones cliente serán desarrolladas bajo el nivel del cliente y así sucesivamente [28]. Para hacer uso de SMDB (Sistemas Manejadores de Bases de Datos), se tendrá que utilizar el API JDBC (*Java DataBase Connectivity*).

3.5.7 Características de la arquitectura J2EE

Para entender y apreciar el J2EE, se deben comprender algunas características que nos permiten ampliar nuestro panorama de lo que en realidad es esta tecnología.

Arquitectura multinivel

Un nivel es una capa de un grupo de componentes de software que ofrece un tipo específico de funciones o servicios. La característica de multinivel forma la base del J2EE. Esto permite que los componentes de software puedan ser distribuidos a través de diferentes computadoras que, a su vez, facilitan la escalabilidad, seguridad y división de tareas durante el proceso de desarrollo y de ejecución.

Como se observa en la figura 3.17, en donde se representa el acceso por multinivel, dependiendo de la aplicación, un cliente puede acceder directamente al EJB o a través de la capa de Web. En principio cualquier cliente puede acceder directamente a la base de datos o a través de un EJB. Sin embargo este proceso depende de la aplicación.

La flexibilidad que presenta la arquitectura permite el desarrollo de un sin fín de aplicaciones con altos grados de complejidad.

Ambiente distribuido

El ambiente distribuido en que el J2EE trabaja permite que diferentes componentes de software sean ejecutados en máquinas diferentes. El trabajo por capas permite a la arquitectura J2EE soportar los ambientes distribuidos.

Es posible ejecutar componentes que pertenecen a todas las capas en una misma máquina o bien en máquinas diferentes, todo dependerá de la función de la aplicación que se esté realizando.

Un ejemplo claro sería tener un servidor dedicado de base de datos y un servidor Web en donde se realice la llamada a los componentes que integran el sistema, ubicados en diferentes computadoras. Uno de tales componentes podrá ser quien verifique si los usuarios que intentan acceder al sistema están o no autorizados, mientras que otro podrá hacer la inserción, la actualización o el borrado de datos.

Portabilidad

La portabilidad en general se refiere a la habilidad de una aplicación para ser ejecutada sobre diversas plataformas. El lema del lenguaje Java es: *escríbelo una sola vez y ejecútalo en cualquier lugar* (debido a que tiene la máquina virtual Java). Esto significa que una aplicación escrita en Java puede ser ejecutada en cualquier sistema operativo, ofreciendo independencia de plataforma.

J2EE ofrece dos dimensiones de portabilidad: independencia de sistema operativo e independencia de desarrollo. El primero es resultado de que el J2EE fue desarrollado basado en la plataforma Java, mientras que el segundo se refiere a que la implementación puede ser desarrollada y ejecutada por cualquier desarrollador.

Interoperabilidad

La interoperabilidad se refiere a la habilidad que el componente de software tiene en interactuar con componentes que han sido desarrollados en diferentes lenguajes de programación. Este punto es muy importante porque las empresas utilizan múltiples plataformas de desarrollo para cubrir sus necesidades.

La interoperabilidad entre múltiples plataformas es vital para integrar sistemas que soportan un desarrollo estándar. J2EE soporta la interoperabilidad con arquitecturas como CORBA (Common Object Request Broker Architecture) y .NET de Microsoft. Como es lógico soporta los protocolos estándar de comunicación: TCP/IP, HTTP y HTTPS.

Escalabilidad

Esta propiedad se refiere a la capacidad de un sistema para crecer sin degradar su funcionamiento. Esto significa que solo hay que agregar funciones adicionales a los componentes y no tener que reprogramarlos por completo.

Disponibilidad y rendimiento

La arquitectura distribuida y multinivel de J2EE soporta un alto rendimiento y disponibilidad. Estas características pueden ser implementadas a través de *cluster* y balanceo de carga. Los servidores pueden ser agregados dinámicamente para satisfacer las demandas de crecimiento de carga.

La arquitectura J2EE comprende diferentes niveles como ya se ha mencionado, y cada nivel contiene componentes, software y contenedores.

3.5.8 Contenedores J2EE

Antes de que un componente sea ejecutado, éste debe ser ensamblado y desarrollado dentro de un contenedor J2EE. En otras palabras una aplicación por componentes puede ser ejecutada solo dentro de un contenedor J2EE. Un contenedor J2EE tiene una aplicación que soporta rutinas que proveen varios tipos de servicios.

Una aplicación cliente, típicamente un programa Java, es empaquetado dentro de un contenedor cliente, y éste se comunica directamente con un contenedor EJB (figura 3.18) o un contenedor Web. El cliente Web, usa el protocolo HTTP para poder comunicarse, exclusivamente se puede comunicar con el contenedor Web. Tanto el contenedor cliente como el contendor Web son implementados por la especificación J2EE.

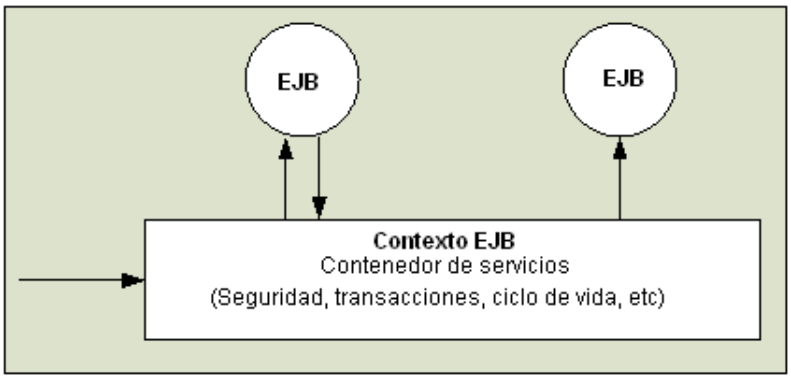

Figura 3.18: Ejemplo de un contenedor EJB

Algunas de las funciones que los contenedores ofrecen son:

- Proveer métodos para que los componentes logren interactuar con otros,
- Proveer APIs que soportan la compatibilidad con ambientes Java,
- Ofrecer servicios para que los componentes puedan administrar transacciones, seguridad, identificación, búsqueda y conectividad remota.

Gracias a estas características, el desarrollo de aplicaciones es mucho más fácil y rápido. Esta arquitectura presenta un ambiente integrado por varios componentes que están a la disposición del programador, por ejemplo JDBC para la conectividad con bases de datos, o el manejo de objetos distribuidos a través de JRMI (Java Remote Method Invocation) para realizar un conjunto de tareas que llegan a conformar y consolidar un solo trabajo en común.

3.5.9 Servicios J2EE

La especificación J2EE requiere de un conjunto de servicios estándar. Estos servicios son utilizados para soportar la aplicación J2EE [35]. Algunos de los servicios más importantes son mencionados a continuación:

- ***HTTP***: define el servicio HTTP donde el API del lado del cliente está definido por el paquete `java.net`. El API del lado del servidor está definido por el API Servlet (incluyendo JSP). HTTPS, o el uso de HTTP sobre "Secure Socket Layer" (SSL), está soportado por los mismos APIs del cliente y del servidor que HTTP.

- ***Enterprise JavaBeans (EJBs)***: un modelo de componentes para el desarrollo empresarial en Java.

- ***Java Transaction API (JTA)***: una interfaz para los componentes de aplicaciones J2EE para manejar transacciones.

- ***Java IDL***: es una tecnología similar a RMI que soporta objetos y componentes distribuidos que han sido escritos en Java. Para soportar la interacción de objetos distribuidos, Java IDL proporciona una implementación del ORB (*Object Request Broker*). Esta implementación permite que se realice la comunicación a bajo nivel entre aplicaciones Java IDL y aplicaciones compatibles con CORBA.

- ***JDBC***: API para acceder a bases de datos.

- ***Java Message Service (JMS)***: la Java Message Service (JMS) API, maneja los modelos: punto a punto y publicación y suscripción. En el modelo punto a punto, un mensaje enviado por un cliente es consumido por otro cliente que lo recibe, mientras que en modelo de publicación-suscripción; un mensaje enviado por un cliente puede ser consumido por múltiples clientes que los reciben.

- ***Java Naming and Directory Interface (JNDI)***: debido a que J2EE ofrece un ambiente distribuido, los componentes y los servicios pueden residir en múltiples máquinas. En consecuencia, un mecanismo de búsqueda de nombres es requerido para localizar a los componentes y a los servicios.

- ***Java API for XML Parsing (JAXP)***: un API estándar para analizar documentos XML. Incluye soporte para SAX y DOM.

- ***J2EE Connector Architecture***: un API estándar para permitir la conectividad de sistemas legados y aplicaciones empresariales no Java.

- ***Java Authentication and Authorization Service (JAAS)***: un API estándar para permitir que los servicios J2EE autentifiquen y fuercen el control de acceso sobre los usuarios.

- ***Servicio de transacciones***: administrar las transacciones para la aplicación de componentes es una tarea importante. Esta tarea se realiza en una plataforma independiente, la cual utiliza la Java Transaction API (JTA).

  Como parte de los beneficios existentes dentro de J2EE para Servicios Web, se incluye la portabilidad, la escalabilidad y la fiabilidad. Este conjunto de características se suman a las ya existentes de este conjunto de especifícaciones para Java.

3.6 Paradigma de componentes en J2EE

La especificación J2EE soporta el paradigma basado en componentes mediante la tecnología de los llamados EJB (Enterprise Java Beans). Esta tecnología brinda el soporte para el desarrollo e implementación de componentes en ambientes distribuidos, que incorporan como base de funcionamiento la tecnología Java.

3.6.1 Enterprise Java Beans

Un EJB es un componente escrito en lenguaje Java, que encapsula la lógica de negocio de una aplicación. La lógica del negocio es el código que cumple con el propósito de la aplicación [33].

En la tecnología Java, existen dos tipos de componentes, los "Java Bean" y los "Enterprise Java Bean", ambos responden a la definición del párrafo anterior. La diferencia esencial entre uno y otro es que el primero no soporta el uso de RMI (Remote Method Invocation), por lo tanto no es posible distribuir componentes Java Bean.

Para los segundos existe el "ambiente de ejecución", es decir un contenedor EJB o contenedor de componentes, mismo que permite al EJB ser ejecutado.

3.6.2 Contenedor EJB

Las instancias de los EJB son ejecutadas dentro del contenedor. Éste no es más que un entorno de ejecución que controla las instancias de un EJB y provee toda la administración de servicios necesarios para el ciclo de vida del componente.

El contenedor de componentes EJB provee una interfaz mediante la cual los componentes pueden comunicar con el mundo exterior. Cualquier petición o respuesta que se realice al EJB tendrá que ser obtenida mediante el contenedor de componentes, pues éste último aisla al EJB de un acceso directo por un cliente.

El contenedor intercepta cualquier invocación que sea realizada por un cliente para asegurar la persistencia y asegurar al cliente que se realice la transacción sobre el

Bean. El contenedor de EJB es también encargado de generar un EJB_Home, para localizar, crear y remover el componente EJB.

La interfaz de contexto del EJB, es provista por el contenedor EJB, encapsulando así la información relevante acerca del ambiente del contenedor como la identidad de un componente EJB, el estado de la transacción o la referencia remota existente al mismo Bean.

3.6.3 Servicios del contener EJB

La función principal del contendor de componentes es envolver a los componentes proporcionándoles así un conjunto de servicios. Comúnmente a este hecho se le conoce como capa de servicios añadidos. Entre los servicios más importantes se encuentran:

- **Manejo de transacciones**: inicio y término de las transacciones asociadas al EJB.
- **Seguridad**: control en el acceso de los métodos asociados al EJB.
- **Concurrencia**: control de la llamada simultanea de un EJB por diferentes clientes.
- **Servicios de red**: comunicación entre el cliente y el servidor del EJB en diferentes máquinas.
- **Gestión de recursos**: gestión automática de múltiples recursos, tales como colas de mensajes o bases de datos.
- **Gestión de mensajes**: manejo del Java Message Service (JMS).
- **Escalabilidad**: posibilidad de construir *clusters* de servidores de aplicaciones con múltiples *host* para realizar balanceo de carga con el simple hecho de agregar *hosts* adicionales.

3.6.4 Funcionamiento de los EJB

El funcionamiento de los EJB fundamentalmente se basa en el trabajo del contenedor. Éste no es más que una aplicación Java, la cual es ejecutada en un servidor que contiene clases e instancias de objetos necesarios para el buen funcionamiento de los EJB.

En la figura 3.19, se aprecia el funcionamiento básico de un EJB. Hay que hacer notar que la primera tarea realizada son las peticiones que un determinado cliente puede realizar al EJB, mediante la comunicación con el servidor que contiene al EJB, el cual puede ser ejecutado en diferentes máquinas virtuales e incluso en *hosts* diferentes.

La comunicación antes mencionada se realiza mediante un intermediario, éste es el contenedor de EJB, el cual proporciona una interfaz llamada `EJBObject`. Cualquier interacción que el cliente desee realizar con el EJB la deberá hacer mediante dicha interfaz.

Como se observa en la figura 3.19, el cliente hace una petición mediante la interfaz, la cual solicita al contenedor de componentes una serie de servicios que permiten la comunicación con el EJB, para así poder realizar la transacción que desea, que en el ejemplo es una transacción en la base de datos.

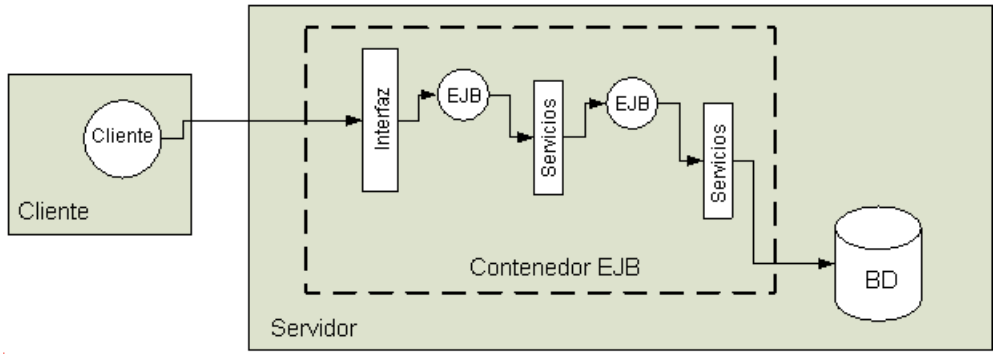

Figura 3.19: Funcionamiento básico de un componente EJB

3.6.5 Modelo de componentes del EJB

Cada componente EJB tiene una interfaz, misma que es publicada para ser utilizada por el cliente, quien solo de este modo puede tener acceso a los métodos del EJB. Para el cliente no es necesario conocer cómo es que esta constituido el EJB, al cliente solo le interesa el resultado que el EJB generará, la interfaz que el cliente maneja se le conoce como *interfaz remota*.

Las instancias de un componente son creadas y administradas por el contenedor mediante una interfaz llamada *interfaz local*. Todos los componentes deben tener una interfaz local y una interfaz remota. Los EJB pueden ser configurados en tiempo de desarrollo.

Así en la figura 3.20 se puede observar como se representa un componente EJB, en donde las líneas punteadas representan al contenedor, lo que se encuentra en su interior es el componente, al cual se puede acceder mediante el uso de sus interfases, lo que se encuentra al exterior es un servidor de aplicaciones J2EE.

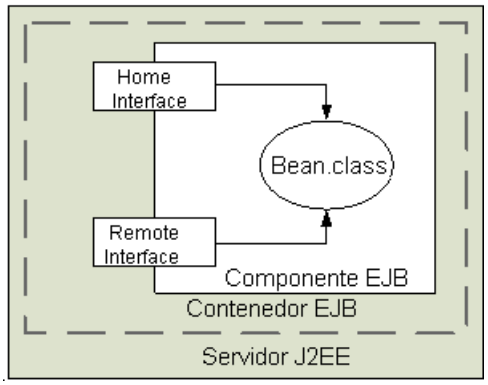

Figura 3.20: Componente EJB

La implementación de un componente EJB se realiza mediante el uso de la interfaz local y remota. Los componentes EJB son de caja negra, es decir, un cliente de un EJB solo conoce lo que el componente hace pero no cómo lo hace.

Un cliente hace una petición a un EJB, el cual es localizado mediante un servicio de nombres y obtiene una referencia al componente. Para realizar la petición el cliente puede crear una instancia del componente en el servidor, de acuerdo a la referencia del componente.

3.6.6 Tipos de EJB's

El modelo de componentes de EJB soporta los siguientes tipos de beans: *beans de sesión, beans de entidad* y *beans dirigidos por mensajes.*

Beans de entidad: representan datos persistentes alojados en una base de datos.

Beans dirigidos por mensajes: pueden escuchar mensajes de un servicio de mensajes de JMS. Estos beans nunca son llamados directamente por los clientes, para realizar la comunicación es necesario enviar un mensaje JMS. Este tipo de bean no necesita objetos `EJBObject` debido a que el cliente nunca realiza comunicación directa con ellos.

Bean de sesión: son beans interactivos, representan procesos o acciones con respecto a la lógica del negocio. Comúnmente, las llamadas a servicios deben comenzar con una llamada al bean de sesión.

3.6.7 Beans de sesión

Este tipo de beans representan sesiones interactivas con uno o varios clientes. Los beans de sesión pueden mantener un estado únicamente mientras dure la interacción

con el cliente, esto significa que el bean de sesión no realiza ningún tipo de almacenamiento de datos y por lo tanto éstos no son persistentes, (ver figura 3.21).

A diferencia de los beans de entidad, los beans de sesión no pueden ser compartidos entre los clientes. Para dar soporte a múltiples clientes es necesario crear una relación uno a uno entre bean y cliente. Existen dos tipos de bean de sesión: **beans de sesión con estado** y **beans de sesión sin estado**.

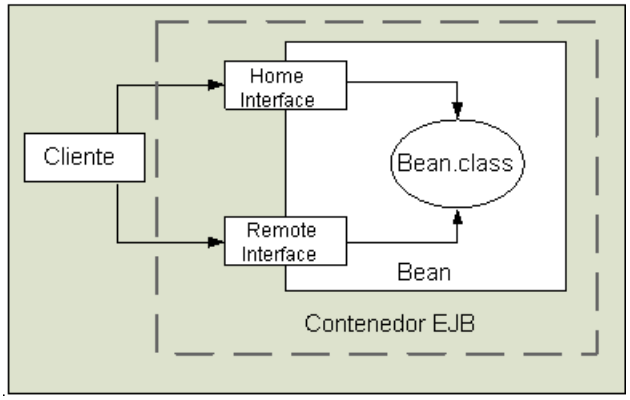

Figura 3.21: Representación de un bean de sesión

3.6.8 Bean de sesión sin estado

Este tipo de bean de sesión no se modifica con las llamadas de los clientes. Cada uno de los métodos que han sido publicados para ser utilizados por los clientes, son llamadas a métodos que reciben datos y retornan resultados. Sin embargo, en este proceso no se modifica la estructura interna del bean, mediante esta metodología se permite al contenedor de componentes crear un *stock* de instancias.

Cada instancia del mismo bean de sesión existe sin mantener un estado. Estas instancias pueden ser asignadas a cualquier cliente o a un conjunto de clientes, ésto es debido a que la asignación solo dura mientras se ha realizado la invocación del método por el cliente.

Cuando un cliente decide realizar la invocación de un método de determinado bean de sesión sin estado, el contenedor EJB obtiene una instancia del *stock*. Debido a que el bean no puede guardar ningún estado, cualquier instancia de éste le sirve al cliente.

En el instante en el cual el método del bean termina su ejecución, la instancia creada es liberada para servir a otro cliente, de tal modo que permite la escalabilidad para un número muy grande de clientes.

3.6.9 Bean de sesión con estado

En este tipo de bean, las variables instanciadas a partir del bean almacenan datos especificos los cuales han sido obtenidos durante la interacción con el cliente. Es decir que cada bean guarda el estado de la interacción que tuvo con el cliente. El estado cambia conforme el cliente realiza transacciones, una vez terminada la sesión el estado es eliminado.

El estado del bean es persistente mientras éste exista, debido a que se tiene que almacenar el estado del bean en cada interacción con el cliente. La administración de éste representa una fuerte carga de trabajo al contenedor de EJB.

Generalmente se deben utilizar beans de sesión con estado cuando se cumplen las siguiente reglas:

1. El estado del bean representa la interacción entre el bean y un cliente específico,
2. El bean necesita mantener información del cliente a lo largo de un conjunto de invocaciones de métodos,
3. El bean hace de intermediario entre el cliente y otros componentes de la aplicación, presentado una vista simplificada al cliente.

3.6.10 Bean de entidad

Este tipo de beans tienen una relación muy directa con la lógica del negocio, debido a que modelan conceptos o datos del negocio y los expresan con nombres. Esta es una regla simple que se sigue para determinar el momento en que se debe implementar un bean de entidad.

Los bean de entidad describen estado y conducta de un objeto que ha sido abstraído del mundo real. El acceso a las variables del bean se realiza mediante los métodos accesores `get` y `set`, y el contenido de sus variables es persistente, es decir que se almacena en una base de datos (ver figura 3.22).

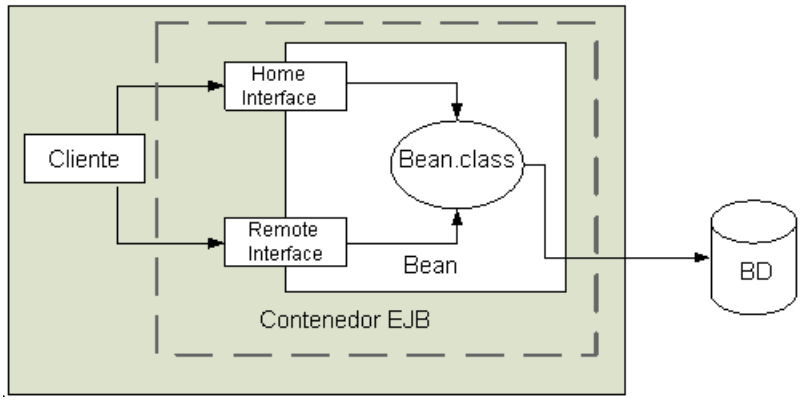

Figura 3.22: Bean de entidad

3.6.11 Beans dirigidos por mensajes

Este tipo de beans, permitente a las aplicaciones J2EE recibir mensaje JMS de forma asíncrona, de tal forma que el hilo de ejecución de un cliente no es bloqueado cuando está esperando que se complete algún método de negocio de otro EJB. Los mensajes pueden ser enviados desde cualquier componente o sistema que no use la tecnología J2EE (ver figura 3.23).

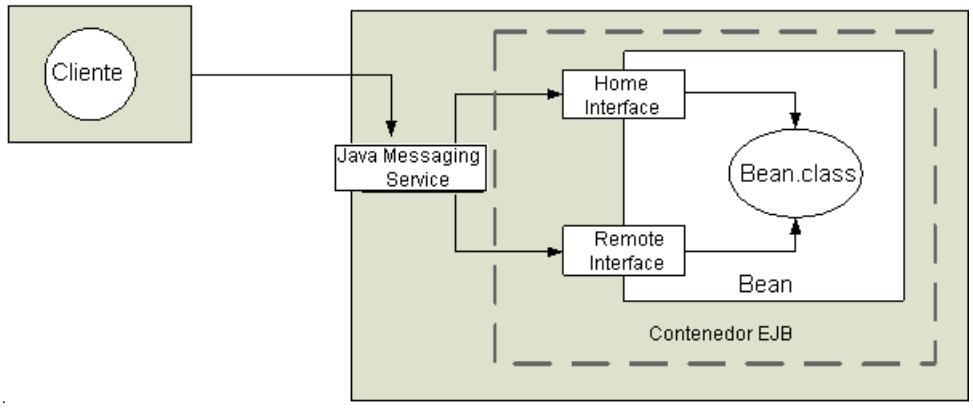

Figura 3.23: Bean dirigido por mensaje

En este capítulo hemos estudiado tres de las herramientas *middleware* más utilizadas en la actualidad, se analizó su arquitectura y el modo en que cada tecnología hace la implementación del *Paradigma Orientado a Componentes* (POC). De lo anterior hemos obtenido que la implementación de componentes en Java se hace mediante los EJB´s, en DCOM con ActiveX y CORBA usa el CCM.

Cada *middleware* tiene una manera propia de realizar la implementación del modelo de componentes, esto representa un problema debido a que no toman un modelo

estándar para realizar la ejecución de componentes, e.g.: Java necesita un contenedor de componentes especifico para EJB que interactúa con la Java Virtual Machine (JVM), DCOM necesita del MTS (que solo se puede ejecutar en plataformas Windows) para poder hacer que los componentes ActiveX funciones y CCM necesita del repositorio de componentes para CORBA, en donde la implementación de CORBA que se esté utilizando debe incorporar soporte para el uso de CCM.

EJB, DCOM y CCM tienen característica que los hacen diferentes uno de otro, éstas particularidades pueden hacer que una herramienta determinada ofrezca mayor o menor robustez en la solución de problemas en la implementación de sistemas distribuidos.

Capítulo 4

Comparativo de herramientas middleware – caso de estudio

En el capítulo anterior hemos estudiando tres arquitecturas diferentes. Se han observado características, estructura básica, modelos de implementación y representación de componentes en cada caso. Esta última característica constituye la piedra angular de nuestro estudio.

En este capítulo, hablaremos de la correlación que existe entre las diferentes arquitecturas *middleware* que hemos estudiado, estableciendo así un marco comparativo entre cada herramienta.

En el capítulo anterior hemos establecido un marco de referencia en cual se hace mención de manera generalizada de las características que cada herramienta *middleware* presenta. Se comenzó por mostrar su arquitectura y modo de operación y posteriormente se estudió la manera en que cada herramienta implementa el paradigma orientado a componentes.

Las herramientas que son objeto de nuestro estudio han sido los siguiente *middleware*: J2EE, CORBA y DCOM los cuales implementan el modelo de componentes mendiante EJB, CCM y ActiveX, respectivamente. A continuación estableceremos un comparativo de las principales características que éstas arquitecturas tienen, así como la diferencia en el modo de implementar el modelo de componentes.

4.1 Administración del ciclo de vida y creación de componentes

En **DCOM**, el servidor MTS (*Microsoft Transaction Server*) es el encargado de administrar, crear y destruir los componentes DCOM, al mismo tiempo se convierte en un contenedor de componentes.

En **EJB** el contenedor de EJB es el encargado de administrar, crear y destruir componentes EJB.

En **CCM** el contenedor de componentes de CORBA es el encargado de administrar, crear y destruir componentes.

Concluimos que cada herramienta necesita de un contenedor para poder implementar y administrar componentes, sin embargo, cada *middleware* tiene una desventaja, es decir, necesita de un contenedor de componentes específico para la ejecución de sus componentes.

4.2 Conectividad a bases de datos

En la mayoría de los sistemas distribuidos, es necesario el uso de una base de datos que permita almacenar datos obtenidos o bien generados por el sistema en determinados momentos. Por ello es necesaria la existencia de mecanismos de gestión de conexiones a bases de datos.

DCOM

La conectividad a base de datos es realizada a través del servicio ODBC (*Open Database Connectivity*). Mediante la utilización de este servicio se pueden gestionar las conexiones a bases de datos, de tal manera que la conexión se facilita ampliamente.

La utilización del ODBC facilita ampliamente la tarea del programador al momento de realizar cualquier transacción sobre la base de datos, debido a que éste solo debe preocuparse de los datos con los que ha de trabajar, ya que la autentificación y el manejo de esquemas es tarea exclusiva del ODBC.

EJB

EJB usa el JDBC (*Java Database Connectivity*) para realizar los accesos a bases de datos. En este caso se deben utilizar manejadores (*drivers*) específicos para la base de datos que se desea administrar. Generalmente estos manejadores son proporcionados

por las compañías desarrolladoras de cada SMBD (*Sistema Maneajador de Base de Datos*).

El EJB hace uso de la API de JDBC que permite administrar las transacciones que se han de realizar dentro de la base datos.

CCM

El acceso a bases de datos, en el caso de CORBA, es un servicio que se realiza mediante el soporte del SMBD (Sistema Manejador de Base de Datos) o bien mediante la implementación que un determinado vendedor haya realizado acerca de CCM. Por ejemplo Postgres ha desarrollado un conjunto de clases que permiten realizar la conexión al cliente mediante el uso de éstas.

En el caso de las implementaciones de CCM para Java, se deja que la implementación gestione la conectividad con la base de datos mediante el JDBC (*Java Database Connectivity*).

4.3 Arquitectura de las herramientas middleware

Las arquitecturas de las herramientas, que son objeto de nuestro estudio, se han descrito en el capítulo 3, de lo que obtenemos lo que sigue.

4.3.1 El modelo DCOM

Los componentes COM son ejecutados bajo el control del MTS. Todos los componentes que han sido desarrollados bajo la estructura de componentes, son almacenados por el registro del sistema operativo Windows. De esta manera los componentes pueden ser implementados por uno o más clientes e incluso por otros componentes.

DCOM se encuentra constituido básicamente por los siguientes elementos:

- El servidor de componentes (MTS),
- El contenedor de componentes (Capa interna del MTS),
- Los servicios inherentes y proporcionados por DCOM. tales como: servicios de ejecución y activación, coordinación de transacciones distribuidas, colas de mensajes, etc,
- Los Clientes DCOM.

4.3.2 El modelo EJB

Los componentes Java son ejecutados y activados bajo el control del contenedor EJB, de tal manera que los componentes EJB pueden ser implementados por uno o más clientes o bien se pueden utilizar para realizar la composición de otros componentes.

Se encuentra constituido por los siguientes elementos:

- Servidor de EJB
- Contenedor de EJB
- Componentes *Home*, *Remote* y los *Java Beans*, que son ejecutadas y activados en conjunto con el contenedor de componentes.
- Clientes EJB
- Servicios proporcionados por el *middleware* J2EE tales como son: servicio de nombres de Java, servicio de transacciones Java, servicios de seguridad, etc.

4.3.3 El modelo CCM

Los componentes CORBA son ejecutados, implementados y activados mediante el contenedor de componentes CCM, De esta manera cada componente puede ser implementado por uno o varios clientes e incluso se pueden usar para la composición de otros componentes.

La arquitectura básica consiste de los siguientes elementos:

- Servidor de CCM
- Contenedor de CCM
- Componentes de Sesión, de Servicio, de Entidad y de Proceso
- Cliente CCM

Hemos llegado a la conclusión de que las tres arquitecturas *middleware* coinciden en la manera de implementar el uso de componentes, es decir, necesitan de un servidor de componentes, de un contenedor de componentes y de la implementación de servicios que serán utilizados por los clientes.

4.4 Soporte de transacciones distribuidas

4.4.1 Modelo de transacciones en DCOM

En el modelo de componentes distribuidos (DCOM), existen primitivas que permiten realizar y administrar las diferentes transacciones que son realizadas con esta herramienta *middleware*. El ambiente de ejecución de MTS automáticamente inicializa las transacciones.

Para todos los componentes, el MTS se encarga de crear la referencia correspondiente a cada componente, la cual contiene información como la identidad de la transacción, el identificador de actividad actual, el identificador del componente cliente que hace la llamada a la transacción, etc.

La referencia es cargada como parte del servidor del componentes. Generalmente cada tarea es ejecutada en máquinas diferentes (cliente – servidor). El MTS utiliza la tecnología OLE para realizar un *commit* en cada fase de la transacción. Para este modelo se definen los siguientes procesos:

1. Transacción cliente
2. Administrador de transacción: coordinador de transacciones distribuidas de Microsoft (MS DTC)
3. Administrador de recursos: cualquier administrador de persistencia, tales como son los sistemas manejadores de base de datos (DBMS).

Para realizar la transacción de manera que ésta sea óptima, el MTS realiza las siguientes tareas:

- Creación y administración del contexto de la transacción
- Instrucción al MS DTC para realizar un `commit` o para abortar la transacción
- Propagación del contexto de la transacción a todos los participantes en la misma para darles a conocer un listado de los recursos disponibles.

4.4.2 Modelo de transacciones para EJB

Los EJB utiliza el servicio de transacciones de Java (JTS) y la API de transacciones de Java (JTA) para el soporte de transacciones. JTA y JTS proveen un conjunto de interfaces y de excepciones que el cliente y los componentes pueden utilizar.

Los elementos de una transacción en EJB son los siguientes componentes:

1. Control
2. Suministro
3. Sincronización
4. Terminación
5. Coordinación

Componente de Control: representa la transacción a realizar. A través de este componente, se puede acceder a los componente de *coordinación* y de *terminación*. El componente *control* es usado por el contenedor EJB para administrar la transacción sobre el Java Bean. Esta tarea es absolutamente transparente para el desarrollador.

Componente de Suministro: contiene el estado de la transacción (éste puede ser, por ejemplo mantener la conexión a una base de datos). Realiza el llamado al `commit()` sobre el componente que realiza la tarea de actualización que será aplicada a la base de datos. De forma similar puede realizar el llamado al `rollback()` sobre el componente que revertirá la transacción.

Componente de Sincronización: este componente necesita ser notificado una vez que la transacción ha finalizado. La transacción se debe hacer en cualquier caso ya sea que transacción fue realizada o revertida.

Componente de Terminación: este componente puede ser usado por el contenedor EJB para realizar la transacción de `commit` o de `rollback`. Un proceso es retornado a un método del Java Bean cuando un `commit` o un `rollback` es requerido. Todos los componentes de la transacción realizan cualquiera de las dos tareas antes mencionadas.

Componente de Coordinación: éste se encarga de coordinar todas las tareas. Los componentes *Suministro* y *Sincronización* son registrados por el *Componente de Coordinación* dentro de la transacción. El Java Bean no puede obtener acceso a este componente de forma directa, por lo que solo puede obtener una referencia al componente.

En general la administración de transacciones envuelve dos posibles capas de distribución: *aplicaciones de múltiples participantes* y *administración de recursos con datos múltiples*. Cada capa debe ser administrada por separado, el coordinador de transacciones es responsable de asegurar el soporte para las transacciones distribuidas.

El director de una transacción es llamado *administrador de transacciones o coordinador de transacciones*. La protección de las transacciones y de recursos como es el caso del manejo de la conexión de una base de datos, es gestionada mediante el *administrador de recursos*.

4.4.3 Modelo de transacciones para CCM

Las transacciones se realizan mediante el uso de OTS (*Object Transaction Service*), éste define procesos de administración de transacciones. El OTS se encarga de asegurar la integridad de las transacciones. Este servicio es muy util en aplicaciones multicapas (en donde la bases de datos no es accedida directamente). La implementación de este servicio se hace de la misma manera que en los EJB's.

Podemos concluir que cada middleware utiliza un administrador de transacciones para asegurar la integridad de las transacciones, de tal forma que éstas siempre sean realizadas. La administración se realiza de forma automática, sin embargo, en algunos casos es posible programarla para realizar tareas de administración muy específicas.

4.5 Servicio de transacciones distribuidas

El servicio de transacciones distribuidas es muy parecido en las tres arquitecturas, para lo cual se tienen que:

- DCOM: utiliza el MSDTC
- EJB: utiliza el JTA y JTS
- CORBA: utiliza el OTS

4.5.1 DCOM-MSDTC

El Coordinador de Transacciones Distribuidas de Microsotf (MSDTC) se encarga de coordinar las transacciones realizadas y ayuda a asegurar que la operación sobre transacción distribuida se realice satisfactoriamente. El contenedor de componente se comunica con el MSDTC usando las interfaces de los componentes.

4.5.2 EJB-JTA y EJB-JTS

EJB provee una plataforma de transacciones para los componentes EJB. Los recursos transaccionales son registrados automáticamente por el servidor de componentes EJB. Los componentes EJB pueden interactuar con la plataforma de transacciones mediante el JTA (API de Transacciones de Java), esta permite que las transacciones tales como `commit` o `rollback` sean realizadas.

JTA es solo una interfaz que está completamente desprovista de cualquier implementación. Ésta define una interface para un servicio de transacciones, que puede ser o no el JTS.

4.5.3 CCM-OTS

CCM utiliza el OTS (*Object Transaction Service*) para asegurar que sus transacciones se ejecuten correctamente, éste se encarga de asegurar que exista integridad de datos y gestiona la compatibilidad con múltiples implementaciones de servicios CORBA.

El OTS define interfaces que permiten que múltiples componentes distribuidos cooperen en orden para proveer atomicidad de procesos. Estas interfaces posibilitan a los componentes que cualquiera realice el `commit` de todos los cambios o bien que realice el `rollbak` de todos, incluso en presencia de fallas.

Concluimos que en **DCOM**, cuando un cliente referencia a un componente, se crea una referencia (*Context Object*). Este *Context Object* provee información para el MTS, el cliente, el MSDTC y otras fuentes de datos, como pueden ser las conexiones de base de datos. La información incluye la "*Configuración transacciones"* para cada componente, quien depende de las propiedades que se le hayan incluido cuando fue desarrollado.

En esencia, el desarrollador implementa la lógica del negocio, pero no tiene que implementar la gestión de las transacciones, por ello este trabajo se le deja al MTS.

De manera similar, en **EJB** las transacciones para un componente EJB son implementadas por el contendor de EJB quien automáticamente implementa servicios de transacción. El desarrollador debe definir la interfaz remota y debe implementar la lógica del negocio.

En **CCM** las transacciones son gestionadas por el contenedor de componentes, por lo que el monitoreo, coordinación y administración de transacciones son tareas propias de CORBA, de esta forma el desarrollador solo debe preocuparse de la lógica del negocio.

4.6 Componentes

4.6.1 DCOM

Los componentes para DCOM tienen las siguientes características:

- Típicamente se ejecutan sobre un cliente.
- Pueden ejecutar transacciones consistentes.
- Pueden actualizar los datos contenidos en una base de datos.
- El ciclo de vida es relativamente corto, es persistente únicamente durante la llamada del cliente.
- Son destruidos tan pronto el proceso entre la petición y la respuesta al cliente es realizado.
- Los componentes no representan datos que tienen que ser almacenados en una base de datos.

4.6.2 EJB

Los componentes para EJB son de dos tipos y tienen las siguientes características:

Bean de sesión

- Se ejecutan sobre un cliente. Una instancia de un bean de sesión es una extensión del cliente que lo crea.
- Ejecutan transacciones consistentes.
- Pueden actualizar los datos contenidos en bases de datos.
- El ciclo de vida es relativamente corto, es persistente únicamente durante la llamada del cliente.
- Son destruidos si el servidor de EJB se daña por alguna razón inesperada. El cliente tiene que establecer una nueva conexión con un nuevo Bean de Sesión para reanudar cualquier proceso.
- No representan datos que tienen que ser almacenados en alguna base de datos.

Bean de Entidad

- Pueden compartir el acceso de múltiples usuarios
- Pueden participar en cualquier transacción

- Presentan datos dentro del modelo de la aplicación
- Son persistentes. Su ciclo de vida es largo y los datos están contenidos en el dominio de la aplicación
- Pueden sobrevivir si el servidor de EJB se daña.

4.6.3 CCM

Los componentes para CCM son de cuatro tipos y tienen las siguientes características:

CCM de sesión

- Se ejecutan sobre un cliente. Una instancia de un CCM de sesión es una extensión del cliente que lo crea.
- Ejecutan transacciones consistentes.
- Pueden actualizar los datos contenidos en bases de datos.
- El ciclo de vida es relativamente corto, es persistente únicamente durante la llamada del cliente.
- Son destruidos si el servidor de CCM se daña por alguna razón inesperada. El cliente tiene que establecer una nueva conexión con un nuevo CCM de Sesión para reanudar cualquier proceso.
- Estos no representan datos que tienen que ser almacenados en alguna base de datos.

CCM de Entidad

- Pueden compartir el acceso de múltiples usuarios.
- Pueden participar en cualquier transacción.
- Presentan datos dentro del modelo de la aplicación.
- Son persistentes. Su ciclo de vida es largo y los datos están contenidos en el dominio de la aplicación.
- Pueden sobrevivir si el servidor de CCM se daña.

CCM de Servicio

- No mantienen ningún estado.
- No contienen ninguna identidad.
- No participa en transacciones.

- No es persistente.
- Permite ser implementado por cualquier número de clientes.

CCM de Proceso

- Mantienen su estado.
- Tienen una identidad.
- Su identidad no es visible de forma automática para los clientes.
- Pueden participar en cualquier transacción.
- Se utiliza para representar procesos del negocio.

Concluimos que DCOM, EJB y CCM tienen una forma muy particular de implementar componentes, mientras que COM necesita solo de la implementación de un componente para realizar todas sus tareas, Java y CORBA (la implementación de componentes en estas herramientas es un poco parecido), necesitan de la implementación de un componente específico para realizar determinadas tareas, e.g. para instanciar un componente es necesario el uso del *home*, en el caso de DCOM no es necesario más que estar presente en el registro del sistema.

4.7 Portabilidad

DCOM

Los componente en DCOM solo pueden interoperar sobre la plataforma Windows y para cualquier producto desarrollado por Microsoft. Este tipo de componentes pueden ser usados con Visual Basic, Visual C++ y Visual Java de Microsoft. Esta portabilidad es gestionada a través del MTS, sin embargo si los componentes son retirados de éste no pueden interoperar.

EJB

La portabilidad de los EJB, se puede dividir en dos caminos:

1. Portabilidad a nivel plataforma
2. Portabilidad entre diferentes contenedores y servidores EJB

Debido a que el desarrollo de EJB se realiza en lenguaje Java, la portabilidad a nivel de plataforma se realiza mediante el uso de la JVM, debido a ésto se supone que cualquier programa Java se ejecuta en cualquier plataforma sin problemas.

Ocurre el mismo caso entre los componentes EJB y los servidores EJB. Para cualquier EJB las interfaces entre el contenedor y el servidor han sido desarrolladas en base a la especificación de EJB de Sun Mycrosistems. Por ello la portabilidad entre los servidores EJB es alta.

CCM

Una aplicación CCM puede ser portada fácilmente de un producto ORB a otro ORB, por supuesto mientras exista otro producto ORB que soporte el mismo lenguaje de programación en el que el componente CORBA fue desarrollado originalmente.

La portabilidad de la aplicación es posible debido a que CCM estandariza sus interfaces, es decir que solo se necesita realizar la conexión a la interfaz correspondiente para realizar una tarea específica por la capa de la aplicación que la necesite. Las diferencias entre las implementaciones de CORBA son mínimas debido a que todas han de seguir la especificación de la OMG.

Nuestra conclusión es que la portabilidad de componentes es uno de las partes más importantes en el POC, sin embargo como hemos observado en las arquitecturas estudiadas, no se cumple al cien por ciento, así pues DCOM solo puede interactuar con componentes COM (independientemente del lenguaje de programación) registrados por el MTS, mismo que solo es ejecutado por Windows.

En el caso de EJB es posible portarlo a cualquier plataforma de sistema operativo (siempre que éste tenga la JVM) y puede ser usado en cualquier contenedor EJB no importando quien sea el proveedor.

CORBA puede ser implementado en cualquier plataforma de sistema operativo y por cualquier lenguaje de programación en donde se tenga un proveedor de servicios CORBA.

4.8 Interoperabilidad

DCOM

Los componentes administrados por el MTS pueden ser invocados por máquinas remotas a través del uso de DCOM o bien máquinas locales con respecto a los componentes COM. El protocolo que se utiliza para realizar invocaciones remotas es el ORPC (*Object Remote Procedure Call*). Para comunicarse a través del HTTP, los clientes pueden establecer una comunicación con el MTS mediante el Microsoft Internet Information Server (*ISS*) quien es el servidor Web construido para Windows.

EJB

Los EJB pueden ser invocados por máquinas remotas a través de la utilización de RMI o usado por máquinas locales, a través del servidor de componentes y del servidor J2EE. El protocolo utilizado para soportar las invocaciones remotas es el JRMP (*Java Remote Method Protocol*). Depende de la implementación de J2EE para realizar la comunicación HTTP, ésto mediante el servidor de aplicaciones Web.

CCM

CORBA es interoperable entre componentes desarrollados en diferentes implementaciones de CORBA-CCM, debido a que CORBA ha estandarizado sus protocolos de comunicación, por lo que un componente CORBA desarrollado en Java podrá ser utilizado por otro componente desarrollado en C.

Desafortunadamente CORBA no ofrece soporte para servicios Web. Sin embargo el soporte de la comunicación se hace mediante el IIOP (*Internet Interoperability Protocol*), el cual permite la interoperabilidad de nodos diferentes mediante el ORB (*Object Request Brokers*)

De este apartado concluimos que cada *middleware* implementa protocolos para realizar la interoperabilidad con los componentes que hayan sido implementados para esa herramienta en específico, esto es posible conseguirlo mediante la implementación de protocolos de comunicación.

4.9 Administración de persistencia

DCOM

En DCOM los componentes no son persistentes, a pesar de que pueden contener información que necesite persistencia. DCOM no soporta de forma directa la persistencia; si esta es necesaria, el desarrollador necesitará implementar su propio algoritmo de control de persistencia.

EJB

EJB tiene dos tipos de componentes: persistentes (bean de entidad) y no persistentes (bean de sesión). Un bean de entidad representa datos persistentes. Cada bean de entidad tiene un ciclo de vida largo. Cada bean de entidad puede ser compartido por múltiples clientes. Los EJB definen especificaciones para implementar y administrar la persistencia dentro del modelo de componentes EJB. Siempre que los componentes

son creados o destruidos o siempre que son cargados o descargados de memoria, J2EE implementa algoritmos de persistencia sobre los componentes.

La administración de persistencia en los beans de entidad se puede dar en dos casos: cuando el bean administra su propia persistencia, ésto es llamado *Administración de persistencia de Bean (APB)* y cuando el bean delega las tareas de administración de persistencia al contenedor de componentes, a ésto se le llama *Administración de persistencia de contenedor (APC)*.

En el caso del APB, el bean tiene que implementar las operaciones de persistencia directamente en los métodos de la clase del bean, usando JDBC. En el caso de APC, el bean delega la administración de persistencia al contenedor EJB.

CCM

La administración de persistencia se hace mediante el PSDL (*Persistent State Definition Langueage*) y el CIDL (*Component Implementation Definition Language*), a través del uso de los servicios de persistencia de CORBA, al igual que en DCOM, el programador puede implementar sus propios algoritmos de persistencia.

El POS (*Persistence Object Service*) o servicio de persistencia, permite a los componentes ser persistentes más allá de la aplicación que lo creo o de los clientes que lo usan. El POS permite almacenar el estado de un componente en un almacén persistente y devolverlo cuando éste sea necesitado.

El componente es responsable de administrar su estado, pero también puede usar el POS o delegar su trabajo a otro componente. Existen una gran variedad de implementaciones del POS. Cada implementación de CORBA tiene una, dependerá del usuario cómo desea utilizar la persistencia y cómo es que ha de implementarla.

De esta sección concluimos que solo Java y CORBA son capaces de administrar de forma automática la persistencia, en el caso de DCOM es necesaria programarla, aspecto que puede debilitar la seguridad de la aplicación con respecto a la información que maneja.

4.10 Reutilización de software

Para el caso de **DCOM**, **EJB** y **CCM**, los tres tienen un alto grado de reusabilidad, debido a que hace uso de la implementación del modelo de componentes. Bajo esta perspectiva los costos de desarrollo de software se reducen ampliamente, pues los componentes pueden ser reutilizados en varias partes del sistema.

La diferencia fundamental entre las tres herramientas *middleware* estriba en:

1. DCOM solo puede ser implementado en Microsoft Windows y sus componentes no pueden ser llevados a otra plataforma como Linux o UNIX. Sin embargo, si pueden ser implementados por varios lenguajes de programación que sean exclusivos de Microsoft.

2. EJB es portable en cuanto a plataformas y es posible implementarlo en cualquier plataforma (sistema operativo) que haga uso de un contenedor J2EE. Sin embargo, los componentes no pueden ser implementados más que por aplicaciones que hayan sido escritas en lenguaje Java.

3. CCM es portable tanto a nivel plataforma como entre lenguajes de programación, siempre que exista la implementación CORBA para un lenguaje determinado, la única condición para poder utilizar este tipo de componentes es que en las plataformas en donde se vayan a utilizar los componentes exista una implementación CORBA.

Concluimos que la reutilización es alta en el caso de las tres herramientas, siempre y que ésta se realice exclusivamente con componentes que hayan sido desarrollados para esa herramienta en específico.

4.11 Escalabilidad

DCOM, EJB y CORBA permiten un alto grado de escalabilidad, debido a que el uso de componentes permite ensamblarlos a través de otros componentes e incluso permite modificar el comportamiento de un determinado componente, con solo agregar una nueva interfaz, que haga la nueva tarea sin tener que modificar el componente de forma directa y tener que intervenir en los comportamientos anteriores del mismo.

Los mecanismos de composición de componentes, o cambio de comportamiento del mismo mediante el uso de nuevas interfaces son el mecanismo mediante el cual, se asegura que la aplicación sea fiable con respecto de los cambios, pues seguirá con su funcionamiento normalmente, solo se adicionarán las nuevas tareas.

Nuestra conclusión en este aspecto estriba en asegurar que es siempre posible escalar un sistema que haya sido diseñado bajo el paradigma de componentes, desafortunadamente la escalabilidad solo versa en diseños específicos de un solo *middleware* con nuevos componentes desarrollados para ese único *middleware*.

4.12 Documentación

La documentación y soporte, tal vez es uno de los principales problemas a los que se puede enfrentar un programador cuando comienza con un nuevo paradigma de programación o bien, en el momento en que empieza a utilizar una nueva herramienta de desarrollo, el caso de las herramientas *middleware* que hemos estudiado no es la excepción.

Para el caso de **DCOM** y **J2EE** (hablando exclusivamente de componentes), existe una variada bibliografía que puede ser obtenida fácilmente por el desarrollador, por ejemplo podemos mencionar algunas publicaciones y sus editoriales: Willian Rubin y Marsahll Brain [34], Richard Monson [3], Ted Pattinson [27], Eric Armstrong [2], Venkata S. R. Paul [28], Micah Silverman [33].

Para el caso de CORBA, la cantidad de documentación disminuye considerablemente, aun cuando se tiene el soporte de la especificación de CCM realizada por la OMG. Cada implementación de CCM para CORBA es un poco diferente con respecto de otras, por ello es necesario obtener el soporte por el desarrollador de la implementación.

En algunos casos el software es gratuito y lo que verdaderamente vende la compañía es el soporte y por supuesto la documentación, y en otros casos tanto el software como la documentación es vendida por el desarrollador.

Aun con lo anterior, existen algunas publicaciones sobre CORBA, por ejemplo Steve Vinoski [25] y Filtan Bolton [4], cubriendo sus características generales y por supuesto la tecnología de componentes. Estas dan un panorama amplio de cómo esta constituida la arquitectura del *middleware* y de algunos ejemplos muy generales que pueden ayudar al desarrollador.

La mayoría de las publicaciones existentes, tratan de forma amplia el funcionamiento de cada especificación. Cada una trae ejemplos prácticos de algunos casos, con los cuales el desarrollador puede trabajar para aprender rápidamente la tecnología que le interese.

Concluimos que solo es posible encontrar gran cantidad de documentación en herramientas *middleware* comerciales, con respecto a los que son libres o gratuitos, la cantidad de documentación disminuye debido a que en ocasiones ésta solo es provista por el desarrollador del sistema.

4.13 Implementación de la especificación

En caso de **DCOM**, la implementación únicamente existe para la plataforma Windows, como un paquete que ya viene con el sistema operativo y existe para todas las versiones de Windows, a partir del NT 4.0 hasta el Windows Vista.

El desarrollo de componentes DCOM, se puede hacer mediante cualquier herramienta, del Visual Studio ó Visual Studio Net. La elección del entorno de desarrollo es decisión del desarrollador.

Para **J2EE**, existen varias implementaciones de esta especificación, por ejemplo tenemos Web Logic, Apache Tomcat, WebSphere y JBoss y para uso didáctico el Application Server Platform Edition de Sun Microsystems, cada una de éstas sirve para el desarrollo de componentes Java (EJB propiamente dicho).

Para **CORBA**, las implementaciones existentes son pocas (hablando exclusivamente de componentes CCM), y en su mayoría existen solo para dos lenguajes de programación: C++ y Java, cada implementación de la especificación de CCM tiene su propia subversión de la especificación, es decir en algunos casos debido a que no ha alcanzado la madurez que requiere la OMG, ha realizado pequeñas adiciones para satisfacer algunas necesidades.

De las implementaciones que se localizaron para realizar este estudio son: CIAO (*Component-Integrated ACE ORB*) [46], MICOCCM(*Mico CORBA Component Model*) [46], OPENCCM (*Open CORBA Component Model*), EJCCM (*Enterprice Java CORBA Component Model*) [45] y FreeCCM [44].

4.13.1 Resultado del análisis comparativo

Se ha establecido una evaluación comparativa de cada herramienta, en torno a las características que cada herramienta presenta, haciendo énfasis que el estudio que se realizo es subyacente a las herramientas *middleware* que soportan el uso de la programación orienta a componentes. En la tabla 4.5, se han plasmado los resultados del análisis que se ha elaborado.

Característica	**EJB**	**DCOM**	**CCM**
Soporte del paradigma orientado a componentes	Si	Si	Si
Soporte de componentes en varios lenguajes de programación	No	Si	Si
Soporte de componentes de *sesión*, *entidad* y *home*	Si	No	Si
Soporte de persistencia mediante componentes	Si	No	Si
Soporte de composición de componentes	Si	Si	Si
Desarrollo de aplicaciones mediante composición de componentes desarrollados en diferentes lenguajes	No	Si	Si
Soporte de escalabilidad y mantenimiento de componentes	Si	Si	Si
Gestión de componentes mediante Componentes *Home*	Si	No	Si
Gestión de componentes mediante contenedor de componentes	Si	Si	Si
Soporte de conexión para base de datos	Si	Si	No
Soporte de IDL para componentes	No	Si	Si
Soporte de transacciones distribuidas	Si	Si	Si
Uso de protocolos de invocación remota	Si	Si	Si
Soporte de Servicios Web	Si	Si	No
Soporte multiplataforma (Sistemas Operativos) de portabilidad	Si	No	Si
Soporte del enfoque empresarial	Si	Si	Si
Soporte de herencia a nivel IDL	No	No	Si
Implementación de la especificación por múltiples vendedores	Si	No	Si
Soporte de interioerabilidad entre componentes	Si	Si	Si
Existencia de documentación y soporte	Bastante	Bastante	Poca

Tabla 4.1: Comparativo de middleware con soporte de componentes

4.14 Ciclo de desarrollo de componentes para una aplicación distribuida

Para realizar el desarrollo de un sistema distribuido basado en componentes de forma debemos:

1. Definir la arquitectura del sistema.
2. Desarrollar cada componente del sistema.

Con cada herramienta que utilizamos en este trabajo se deben seguir algunos pasos específicos para el desarrollo de aplicaciones, como se indica en las secciones: 4.14.1, 4.14.2 y 4.14.3.

4.14.1 DCOM

1. Crear la interfaz del componente.
2. Registrar esta interfaz en la biblioteca de componentes.
3. Implementar el componente mediante su interfaz como cliente y/o como servidor.
4. Compilar el componente cliente.
5. Compilar el componente servidor.
6. Registrar el componente en el MTS.
7. Compilar el componente cliente en conjunto con la biblioteca *Microsoft Active Server Pages Object Library* para que pueda ser utilizado por una página ASP.
8. Implementar el componente dentro de la página ASP deseada.

4.14.2 J2EE-EJB

1. Crear la interfaz *Home*.
2. Crear la interfaz Remota.
3. Crear el componente EJB.
4. Compilar la interfaz *Home*, Remota y el componente EJB.
5. Registrar el componente en el contenedor de componentes EJB.
6. Registrar el componente en el servidor de aplicaciones J2EE.
7. Implementación el EJB dentro de la página JSP.
8. Registrar la página JSP en el servidor de componentes Web J2EE.

4.14.3 CORBA-CCM

1. Crear las interfaces IDL - CCM.
2. Generar los *stubs* y los *skeleton* de la aplicación.
3. Crear la implementación del componente CCM.
4. Compilar el componente cliente CCM.
5. Compilar el componente servidor CCM.
6. Registrar los componentes cliente y el servidor contenedor de componentes.
7. Activar el servidor de componentes CORBA.

Implementación de CCM en páginas JSP

Para lograr la interacción de componentes CCM con páginas JSP se necesita lo siguiente:

1. Crear un Java Bean que interactúe con la página JSP, mediante los métodos accesores `Get` y `Set`.
2. Dentro del Java Bean se deben definir métodos que permitan interactuar con el `shell` del sistema, mediante la clase `PipeInputStreamToOutputStream` de `Runnable` de Java. De esta manera es posible ejecutar los componentes CCM desde el shell.
3. El componente CORBA recibirá como parámetros: el tipo de operación que se desea ejecutar, la dirección IP y el puerto del servidor CCM al que debe dirigirse.

4.15 Caso de estudio: un sistema de ventas

A continuación se presenta un caso de estudio, también se hace mención de las herramientas que han sido elegidas para su desarrollo.

El modelado del problema desarrollado se ha realizado en base a la representación que cada herramienta *middleware* que se utiliza. Se muestran los casos de uso más generales para la situación que se presenta y los diagramas de colaboración y secuencia.

El resultado de las interfaces gráficas será mostrado en forma de ventana, dejando de lado su codificación debido a que no forma parte de nuestro estudio.

Se ha elegido un sistema de ventas, como caso de estudio para desarrollar en cada una de las herramientas *middleware* que se han elegido. Para ello se ha decidido utilizar las siguientes implementaciones:

Middleware	**Implementación**
J2EE – EJB	Application Server PE 8.2 de Sun Microsystems
DCOM	Visual Basic 6.0 de Microsoft y MTS para Windows XP
CORBA - CCM	MICOCCM (Exclusivo para lenguaje C++)

Tabla 4.2: Implementación de los middleware

4.16 Enunciado del caso de estudio

El problema que a continuación se enuncia, ha sido obtenido de Luque [20], debido a que se ajusta a un caso de la vida real y el cual cubre con nuestras expectativas para poner a prueba cada herramienta *middleware*.

Una empresa internacional dedicada a la comercialización de comidas rápidas, tiene la intención de ubicar una franquicia en nuestra localidad, siéndole necesario automatizar la gestión del negocio adaptándolo a las características propias de la clientela esperada. Esta empresa se dedica a la venta de pizzas y bocadillos, además de productos complementarios como refrescos, helados, etc.

Supuestos del problema

SUPUESTO 1: tanto las pizzas como los bocadillos pueden condimentarse con un número de ingredientes de entre un conjunto de ellos con los que trabaja la empresa.

SUPUESTO 2: los ingredientes con los que se hacen los bocadillos pueden ser iguales o distintos a aquellos con los que se hacen las pizzas.

SUPUESTO 3: el número de ingredientes que intervienen en un artículo que se vende (pizza o bocadillo) no está delimitado, pudiendo realizarse una venta de estos artículos sin ningún ingrediente. Es decir, la empresa vende también las bases de las pizzas y el pan con los que prepara las pizzas y los bocadillos, respectivamente.

SUPUESTO 4: cada artículo que vende la empresa (pizzas, bocadillos y productos complementarios) tiene un precio base asignado (el precio de estos productos cuando se venden de forma independiente), sin contar los ingredientes que pueden acompañar a alguno de estos tipos de artículos.

SUPUESTO 5: cada ingrediente tiene un precio para los bocadillos, mientras que para las pizzas todos los ingredientes tienen el mismo precio (los productos complementarios no llevan ingredientes).

SUPUESTO 6: los artículos se pueden vender en distinto tamaño, en cuyo caso el precio base es distinto según el tamaño, y el precio de los ingredientes también. Existen, actualmente, tres tamaños en los que se venden los bocadillos y las pizzas (pequeño, normal y grande).

SUPUESTO 7: las ventas se pueden hacer de tres formas diferentes: para consumir en el local, para recoger en el local y llevar o consumir en el mismo, y para servir a domicilio, en cuyo caso puede incrementarse un cargo añadido por el porte de la venta, cualquiera de estos procesos se realiza mediante puntos de venta.

SUPUESTO 8: los artículos complementarios que vende la empresa tienen un precio fijo en la base a su tipo, tamaño, sabor, etc.

SUPUESTO 9: los clientes pueden solicitar un servicio de la empresa (un pedido) tanto personalmente en el local como telefónicamente.

SUPUESTO 10: en los pedidos telefónicos se tomarán los datos completos del cliente, y en los de consumir en el local no se tomará ningún dato, a no ser que sea necesario por otras razones.

SUPUESTO 11: los precios de todos los artículos que vende la empresa tienen el IVA incluido.

SUPUESTO 12: la empresa cuenta con una serie de pizzas y bocadillos *estrellas*, los cuales están formados por un conjunto de ingredientes predeterminados. Estos artículos tienen un nombre comercial único y su precio es el que resulta del acumulado de los ingredientes que incorporan.

4.17 Arquitectura general del sistema

Para resolver el problema del sistema propuesto se ha modelado la siguiente arquitectura (ver figura 4.1), en donde observamos un conjunto de servidores en los cuales se encuentran distribuidos un conjunto de componentes. Cada componente tiene asignada una tarea específica, misma que resolverá un problema específico en el problema de las ventas.

Así por ejemplo un servidor contendrá el componente específico para realizar una venta, mientras otro contendrá el componente que se encargue de realizar el alta de los artículos y los precios de los mismos, que la empresa vende.

Los servidores se conectarán a una base de datos, misma en donde se encontrarán todos los registros necesarios para realizar ventas, altas, bajas o modificaciones, según lo crea conveniente la empresa.

La comunicación con el sistema se realizará mediante un portal al cuál se podrán conectar N clientes que soliciten un servicio (puntos de venta), este portal se conecta mediante una red, con los servidores de componentes.

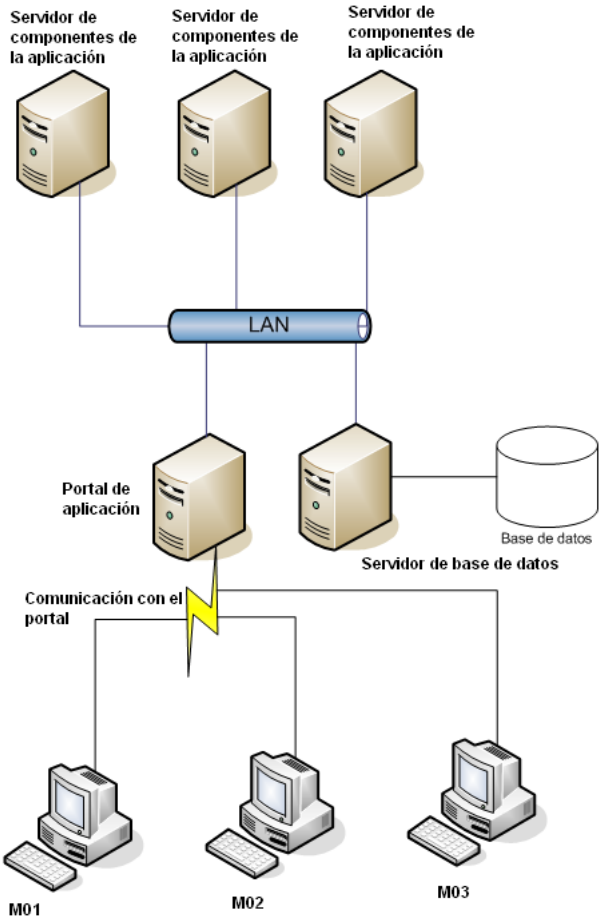

Figura 4.1: Arquitectura general del sistema de ventas

4.18 Diagramas de casos de uso

Para el desarrollo del sistema propuesto, se han tomado en cuenta solo dos casos de uso muy generales, los cuales son suficientes para los efectos que se pretenden.

Para ello las figuras 4.2 y 4.3, representan el caso de uso para un Vendedor y para el Administrador del sistema.

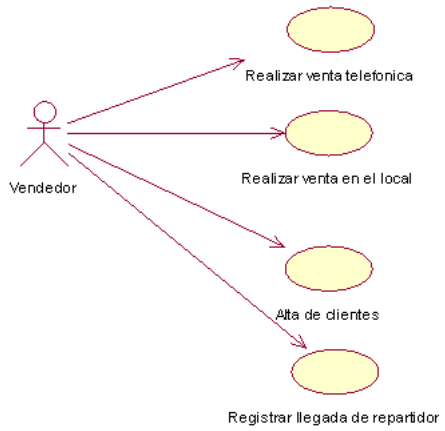

Figura 4.2: Diagrama de caso de uso para Vendedor

Tabla de actividades de la figura 4.2:

Figura UML	**Descripción**	**Servicio proporcionado**
Actor: Vendedor	Se encarga de realizar cualquier tipo de venta, tanto en el local como vía telefónica, dar de alta a los clientes en el sistema. En el caso de que un repartidor haya concluido con su entrega, el vendedor será notificado para registrar su llegada en el sistema.	• Realizar ventas telefónicas. • Realizar ventas en el local. • Realizar el alta de clientes. • Registrar la llegada de repartidores.

Tabla 4.3: Actividades para el Vendedor

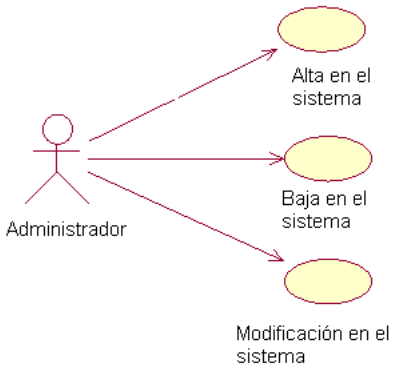

Figura 4.3: Diagrama de caso de uso para Administrador

Tabla de actividades de la figura 4.3:

Figura UML	**Descripción**	**Servicio proporcionado**
Actor: Administrador	Se encarga de realizar cualquier alta, baja o modificación que sea necesaria para la empresa y que tiene que verse reflejado en el sistema.	• Altas en el sistema • Bajas en el sistema • Modificaciones en el sistema

Tabla 4.4: Actividades para el Administrador

4.19 Diagramas de componentes

En las siguientes páginas se presentaran los diagramas de componentes, correspondientes a DCOM, J2EE y CORBA. Cada diagrama representa la manera en que interactúan los diferentes componentes de cada tecnología, para realizar una tarea específica. Hay que notar que mientras en la POO se utilizan diagramas de clase en la POC se utilizan diagramas de componentes.

4.19.1 DCOM

Diagrama de Componentes

DCOM, es una de las herramientas más sencillas a utilizar, sin embargo no por ello se debe dejar de lado la labor de análisis del problema. En la figura 4.1, se muestra la arquitectura genérica que el sistema debe de tener. A partir de esta arquitectura podemos comenzar con el diseño y la organización a base de componentes.

Para ello hemos trasladado ese diseño genérico, a un diseño de componentes específico, el cual pueda ser desarrollado e implementado mediante la herramienta DCOM, tal y como se puede apreciar en la figura 4.4.

El diagrama de componentes que se elaboró, consta de varios elementos:

1. **Servidores de aplicación (nodo servidor)**: cada servidor, contiene un conjunto de componentes, mismos que sirven para que la aplicación realice determinada tarea en cualquier momento en cada servidor. Cada componente realiza una tarea específica. Dentro de los servidores también se encuentra un servidor de componentes (MTS) y un servidor Web (IIS). El servidor de componentes representa a un nodo del sistema distribuido y en este esquema se representa mediante un rectángulo que contiene los elementos ya mencionados (ver figura 4.5).

2. **Servidor de componentes**: este se encarga de administrar la activación y el ciclo de vida de cada componente, así como permitir la interacción con componentes externos o internos. El servidor de componentes se encuentra representado mediante líneas punteadas y contiene los componentes de la aplicación (ver figura 4.5).

3. **Componentes**: cada servidor contiene un número de componentes, mismos que realizan una tarea específica. En el diagrama se pueden observar los siguientes componentes: `COMLista`, `COMListaCom`, `COMTabla`, `COMConsulta` y `COMInsertar`. Todos estos componentes se encuentran dentro de un *servidor local*, sin embargo los cuatro primeros se encuentran en un *servidor In-process*. Cada componente ha sido desarrollado en Visual Basic 6.0 (ver figura 4.5).

4. **Servidor Web**: dentro del nodo servidor se encuentra un servidor Web, en este caso es el *IIS* para Windows. Este servidor contiene un conjunto de *páginas ASP*, mismas que interactúan con los componentes del sistema para realizar una tarea específica (ver figura 4.5).

5. **Portal Web**: el portal Web (ver figura 4.4), contiene la ubicación de cada uno de los nodos, que contienen a los componentes de la aplicación. El portal es el encargado de establecer la comunicación entre los diferentes clientes y los nodos servidor del sistema.

6. **Base de datos**: ésta se encuentra en un servidor dedicado, el cual contiene una base de datos relacional, administrada en Postgres y que contiene la información referente al sistema de ventas (ver figura 4.4).

7. **Clientes**: utilizan el sistema, para ellos la comunicación y la interacción con el sistema es totalmente transparente (ver figura 4.5).

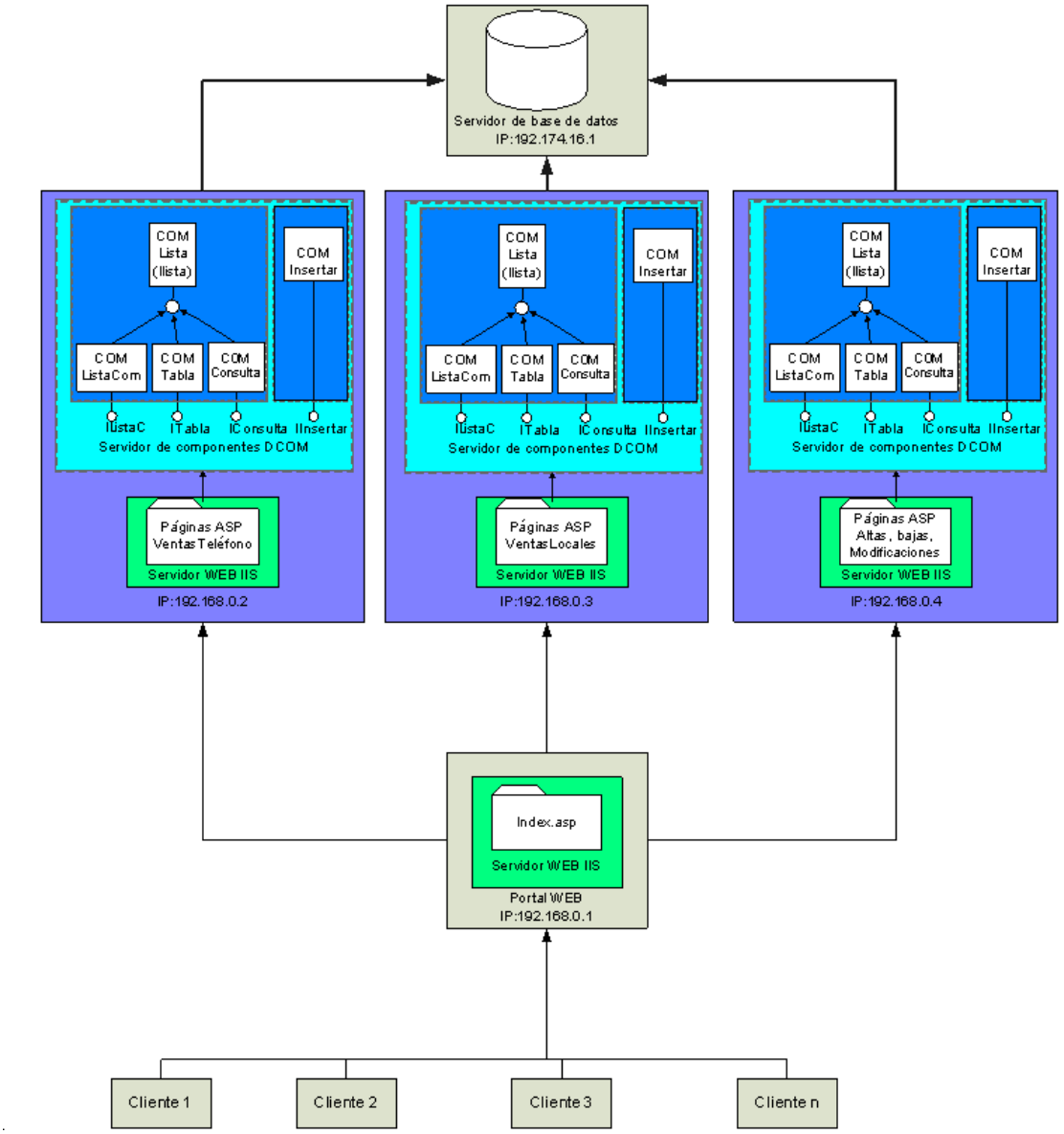

Figura 4.4: Diagrama de componentes DCOM

En la tabla 4.5 se hace una descripción de la figura 4.4.

Elemento	**Descripción**	**Servicio proporcionado**
Componente: COM Lista	Se encarga de mostrar una lista en base a una consulta determinada.	Muestra una lista, de acuerdo a una consulta determina, realizada por un componente.
Componente: ListaCom	Genera una lista desplegable en una página HTML.	Se vincula con COM Lista, para generar una lista de elementos, en base a una consulta.
Componente: Tabla	Genera una tabla para ser desplegada en una página HTML.	Se vincula con COM Lista, para generar una tabla de elementos, en base a una consulta.
Componente: Consulta	Realiza una consulta determinada.	Se vincula con COM Lista para generar un resultado en base a la consulta.
Componente: Insertar	Se encarga de insertar un elemento en la base de datos.	Inserción en la base de datos.
Contenedor de componentes (ver figura 4.5)	Contiene al conjunto de componentes de un modulo o subsistema.	Administra el ciclo de vida de los componentes.
Servidor de componentes (ver figura 4.5)	Contiene a uno o varios contenedores de componentes.	Permite interactuar con el exterior a todos los componentes que se encuentren dentro de éste.
Nodo Servidor	Contiene al servidor de componentes, así como al conjunto de aplicaciones, módulos o submodelos del sistema.	Permite que otros nodos se conecten a él a través de una red.
Servidor Web IIS	Contiene las páginas ASP que formar parte de la interfaz de usuario del sistema.	Gestiona peticiones y respuestas vía *http*.
Portal Web	Ofrece al usuario de forma fácil e integrada, el acceso a los componentes y sus servicios	Se encarga de gestionar el acceso al sistema distribuido, para que éste sea transparente al usuario.
Clientes	Se encargan de conectarse al sistema para realizar alguna transacción.	Realiza transacciones como son: ventas, altas, bajas y modificaciones.

Tabla 4.5: Descripción de la figura 4.4

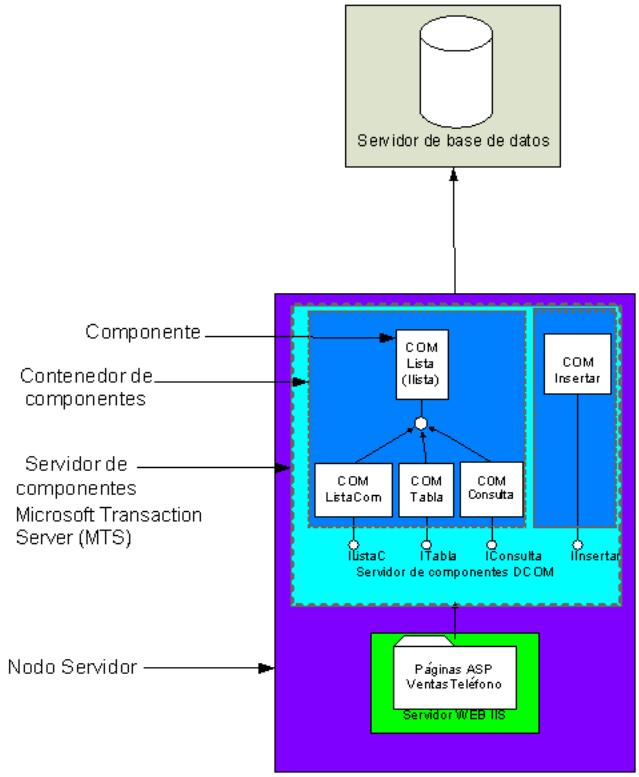

Figura 4.5: Diagrama de un nodo de componentes DCOM

4.19.2 J2EE-EJB

Diagrama de Componentes

J2EE es una herramienta muy robusta para el desarrollo de sistemas empresariales. Dentro de su conjunto de especificaciones, cuenta con un gran número de API's que facilitan el trabajo del desarrollador. Sin embargo, no hay que dejar de lado el diseño del sistema.

El modelo mostrado en la figura 4.1, lo hemos traslado a un modelo de componentes en la terminología de EJB, ver figuras 4.6 y 4.7. El modelo que presentamos es muy parecido al que se utiliza en DCOM.

El diagrama de componentes para EJB consta de los siguientes elementos:

1. **Servidores de aplicación (nodo servidor)**: en cada servidor se encuentran los componentes de la aplicación, cada uno de éstos realiza un conjunto de tareas específicas. Dentro de los nodos servidor se encuentra un Servidor Web y un Servidor de aplicaciones J2EE, y son implementados en el `Application Server PE 8.2` de Sun Microsystems. El nodo servidor se encuentra representado por un rectángulo el cual contiene los elementos antes mencionados.

2. **Servidor de componentes EJB**: es el encargado de administrar la activación y ejecución de los componentes EJB. Mediante el uso del *Home*, también permite la interacción entre componentes remotos mediante el servicio de nombres.

3. **Servidor de componentes Web**: este servidor contiene un conjunto de *páginas JSP* que sirven como interfaz del sistema y al mismo tiempo hace una composición de componentes con los componentes provistos por el Servidor de componentes EJB. Finalmente se tiene un componente que realiza una tarea específica, como es el caso de `VentasTelefono`, `VentasLocales`, `Altas`, `Bajas` y `Modificaciones`, administradas por `BeanListar`.

4. **Servidor Web**: en el nodo servidor se encuentra un servidor Web que soporta la comunicación con los otros servidores con la base de datos. Este servidor Web al mismo tiempo es un Servidor de aplicaciones J2EE, por ello contiene dentro de sí, a los servidores de componentes EJB y componentes Web, que a su vez contienen a los componentes EJB y a las páginas JSP respectivamente.

5. **Portal Web**: el portal Web, permite que los clientes del sistema se comuniquen con cada uno de los servidores, esto según la tarea que se necesite realizar, éste portal contiene la ubicación de cada uno de los nodos servidor.

6. **Base de datos**: ésta se encuentra en un servidor dedicado, el cual contiene una base de datos relacional, administrada en Postgres y contiene información referente al sistema de ventas.

7. **Clientes**: utilizan el sistema, para ellos la comunicación y la interacción con el sistema es totalmente transparente.

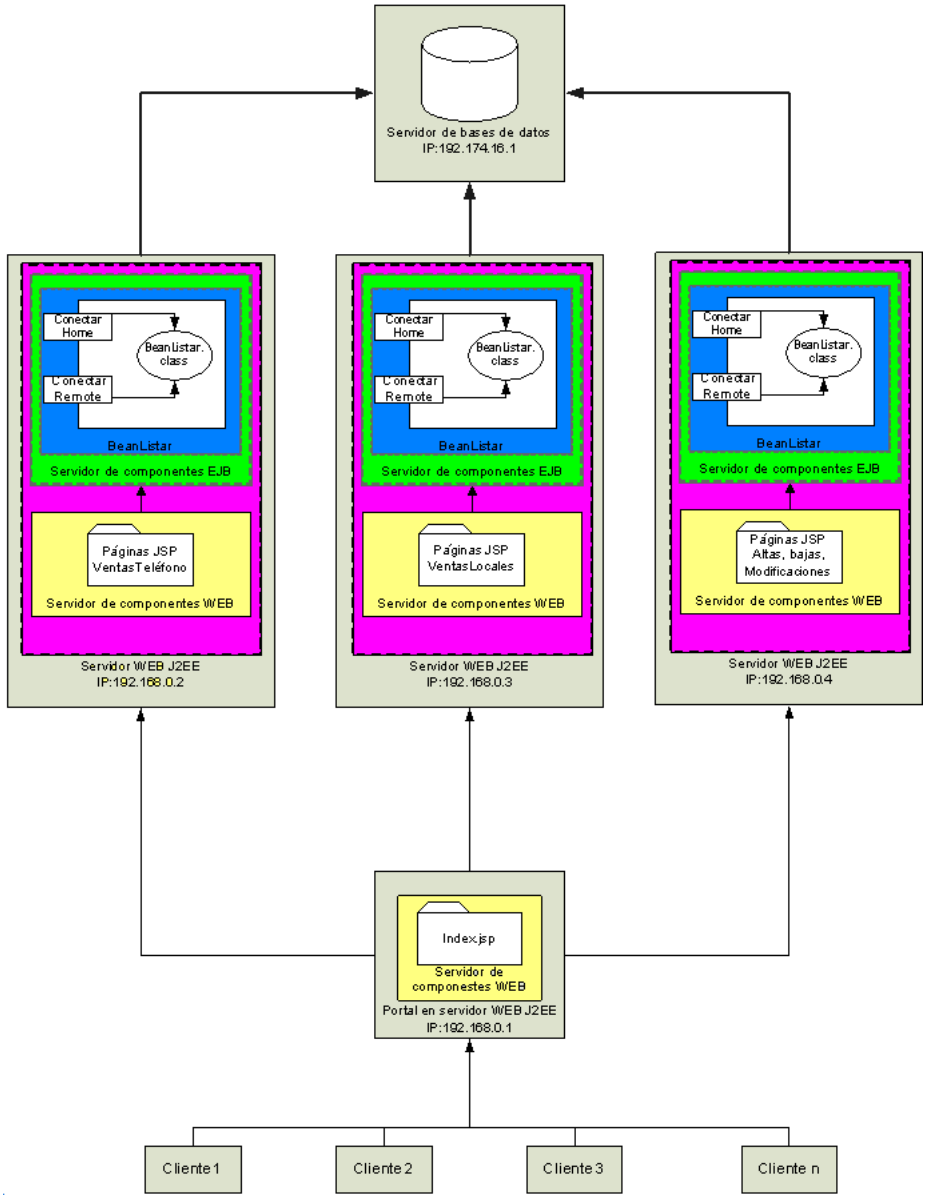

Figura 4.6: Diagrama de componentes J2EE

En la tabla 4.6 se hace una descripción de la figura 4.6.

Elemento	**Descripción**	**Servicio proporcionado**
Componente: Bean Listar	Es el componente principal, el cual contiene los componentes *Home* y *Remote*.	Se encarga de gestionar algunas actividades, como pueden ser ventas, altas, bajas o modificaciones.
Componente: ConectarHome	Es la interfaz del componente *BeanListar*.	Permite interactuar a *BeanListar* con el mundo exterior.
Contenedor de componentes (ver figura 4.7)	Contiene al conjunto de componentes de un módulo o subsistema.	Administra el ciclo de vida de los componentes.
Servidor de componentes (ver figura 4.7)	Contiene a uno o varios contenedores de componentes.	Permite interactuar con el exterior a todos los componentes que se encuentren dentro del sistema.
Nodo Servidor	Contiene al servidor de componentes, así como al conjunto de aplicaciones, módulos o submódulos del sistema.	Permite que otros nodos se conecten a él a través de la red.
Servidor Web J2EE	Contiene las páginas JSP que formar parte de la interfaz de usuario del sistema, así como la aplicación misma J2EE.	Gestiona y administra la interaccion de los componentes Java, en un entorno *on-line* o *stand alone* de la aplicación.
Portal Web	Ofrece al usuario de forma fácil e integrada, el acceso a los componentes y sus servicios	Se encarga de gestionar el acceso al sistema distribuido, para que este sea transparente al usuario.
Clientes	Se encargan de conectarse al sistema para realizar alguna transacción.	Realiza transacciones como son: ventas, altas, bajas y modificaciones.

Tabla 4.6: Descripción de la figura 4.6

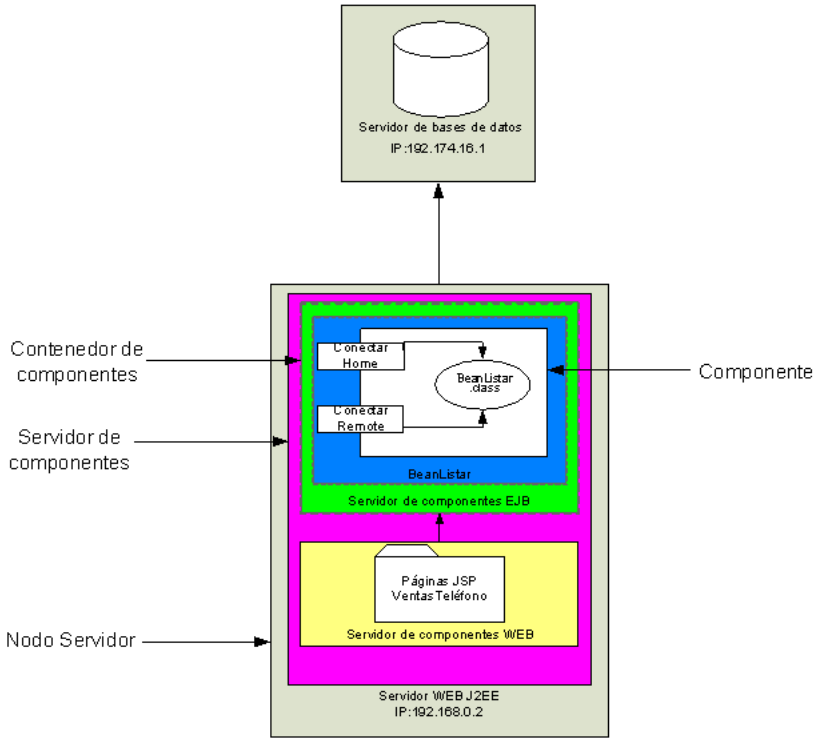

Figura 4.7: Diagrama de componentes J2EE

4.19.3 CORBA-CCM

Diagrama de Componentes

El diseño de componentes para CORBA-CCM es mucho más complejo y abstracto de lo que pareciera, debido al grado de acoplamiento que se debe de lograr en el desarrollo de los componentes.

Para tener claro el modelo de componentes que se debe de diseñar es necesario comprender claramente la arquitectura genérica del sistema, ver figura 4.1. Una vez que se tiene bien comprendida es posible modelar un diagrama de componentes para CCM, ver figura 4.8.

El diagrama de componentes para CCM consta de los siguientes elementos:

1. **Servidores de aplicación (nodo servidor)**: cada servidor contiene los componentes de la aplicación, un servidor de componentes, un contenedor de componentes. La implementación de estos se ha realizado en MICO [11]. El nodo servidor se encuentra representado por un rectángulo (ver figura 4.8).

2. **Servidor de componentes CCM**: éste es el encargado de administrar la activación y ejecución de los componentes CCM, mediante el uso del *Home* (ver figura 4.9). También permite la interacción entre componentes remotos mediante el servicio de nombres.

3. **Contenedor de componentes**: contiene todos los componentes cliente y servidor que han sido creados para realizar determinada tarea. El componente servidor contiene a `CCM_HOME` y a `CCM_IMP`. Estos últimos responden a los eventos generados por el componente cliente, ver figura 4.9. El contenedor de componentes es el encargado de comunicar a los componentes cliente y servidor entre sí y con el exterior, coordinándose siempre con el servidor de componentes (ver figura 4.8).

4. **ORB**: es el encargado de establecer la comunicación entre los clientes y servidores CORBA.

5. **Servidor CORBA**: se comunica con el ORB para establecer la comunicación con otro servidor CORBA o bien con un cliente CORBA (ver figura 4.8).

6. **Cliente CORBA**: se comunica con el ORB para establecer la comunicación con los servidores CORBA y también establece la comunicación con el servidor de aplicaciones Tomcat 5.5 (ver figura 4.8).

7. **Portal Web**: contiene a un cliente CORBA y a un Servidor de Aplicaciones J2EE. Este contiene un conjunto de páginas JSP que permiten a los clientes del sistema comunicarse con cada uno de los servidores de la aplicación debido a que contiene la ubicación de cada uno de los nodos servidor. También contiene un conjunto de métodos que permiten que se realice la comunicación con el cliente CORBA y éste a su vez, se comunique con el ORB y los servidores CORBA para realizar una tarea determinada.

8. **Base de datos**: ésta se encuentra en un servidor dedicado, el cual contiene una base de datos relacional, administrada en Postgres y contiene información referente al sistema de ventas (ver figura 4.8).

9. **Clientes**: utilizan el sistema, para ellos la comunicación y la interacción con el sistema es totalmente transparente (ver figura 4.8).

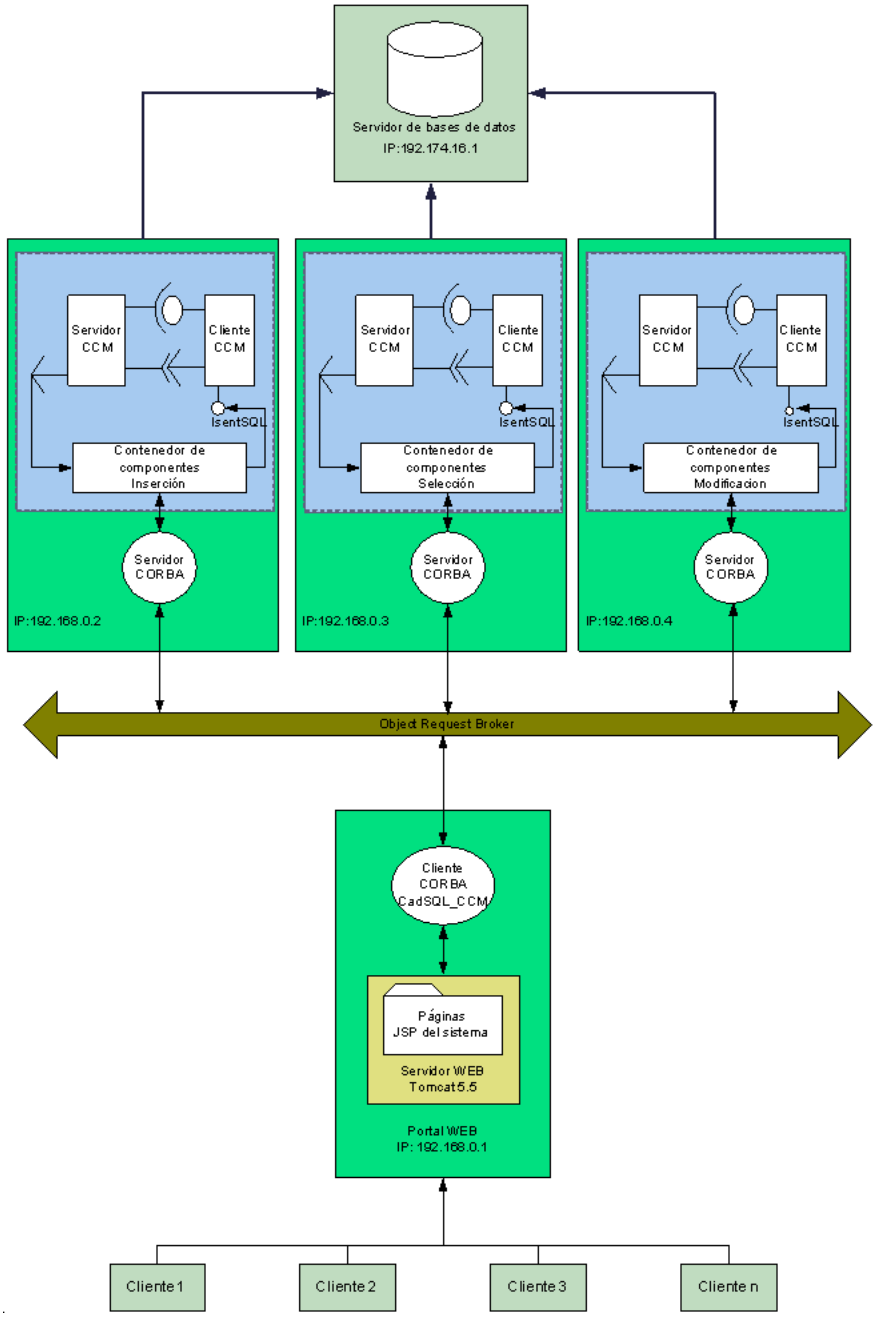

Figura 4.8: Diagrama de componentes CCM para el sistema

En la tabla 4.7 se hace una descripción de la figura 4.8.

Elemento	**Descripción**	**Servicio proporcionado**
Componente: CCM Servidor	Responde a peticiones de un componente cliente.	Responde a peticiones de un componente cliente. Estas peticiones son retornadas al contenedor de componentes como respuesta a un evento generado.
Componente: CCM Cliente	Realiza peticiones a un componente servidor.	Realiza peticiones a un componente servidor, como respuesta al evento generado por el contenedor de componentes.
Contenedor de componentes (ver figura 4.9)	Contiene dentro de sí al conjunto de componentes de un módulo o subsistema.	Administra el ciclo de vida de los componentes.
Servidor de componentes (ver figura 4.9)	Contiene a uno o varios contenedores de componentes.	Permite interactuar con el exterior a todos los componentes que se encuentren dentro del sistema.
Nodo Servidor	Contiene al servidor de componentes, así como al conjunto de aplicaciones, módulos o submodelos del sistema.	Permite que otros nodos se conecten a él a través de una red.
Cliente CORBA	Realiza peticiones a determinado servidor, mediante la comunicación realizada por el ORB.	Comunicarse con el servidor.
Servidor CORBA	Gestiona respuestas a un cliente determinado, mediante la comunicación realizada por el ORB	Comunicarse con el cliente.
ORB	Comunicar a los diferentes clientes y servidores CORBA	Gestionar la comunicación de clientes y servidores CORBA a través de la red.
Portal Web	Ofrece al usuario de forma fácil e integrada, el acceso a los componentes y sus servicios	Se encarga de gestionar el acceso al sistema distribuido, para que éste sea transparente al usuario.
Clientes	Se encargan de conectarse al sistema para realizar alguna transacción.	Realiza transacciones como son: ventas, altas, bajas y modificaciones.

Tabla 4.7: Descripción de la figura 4.8

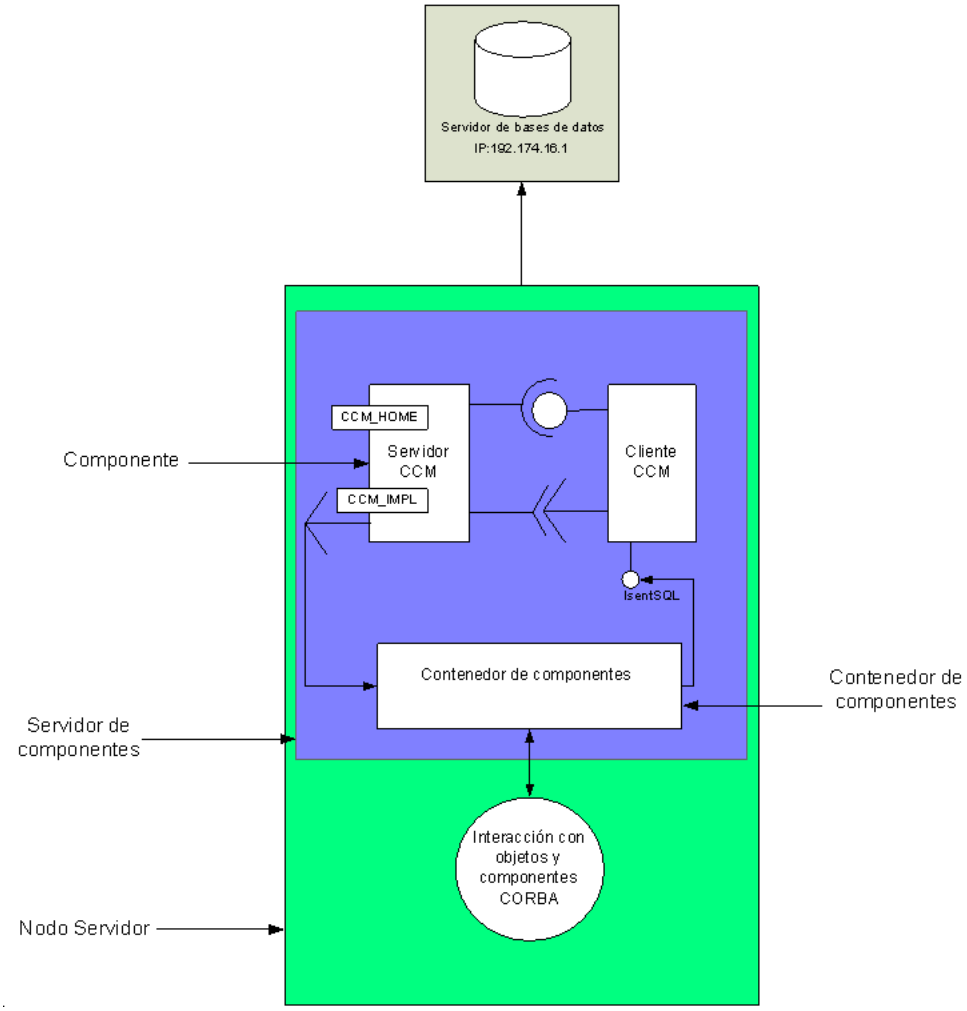

Figura 4.9: Diagrama de componentes CCM

4.20 Diagramas de secuencia y colaboración genericos de la aplicación

En cuanto a los esquemas complementarios, como son los de colaboración y secuencia, se sigue la misma simbología UML para ambos paradigmas. Debido a que uno es consecuencia del otro se han puesto en el siguiente orden: primero el diagrama de secuencia y posteriormente el diagrama de colaboración. Éste último debe de leerse en base a la secuencia numérica que sigue, así por ejemplo de la actividad 1, seguirá la actividad 2 y así sucesivamente.

Este conjunto de actividades guardan una correspondencia directa con el diagrama de secuencia. Cabe aclarar que debido a la similitud de actividades que se siguen en el sistema, hemos desarrollado diagramas genéricos que sirven para el desarrollo de la aplicación con cualquiera de las herramientas *middleware* que estamos analizando.

4.20.1 Diagrama de colaboración y secuencia - ventas locales

En el diagrama de la figura 4.10 se muestra como el vendedor interactúa con el sistema para realizar una venta. Según la tarea que desee realizar, se activa una serie de componentes.

1. **Realizar Venta:** cuando se requiere realizar una venta, el vendedor tiene que decir al sistema que tipo de venta debe realizar. En este caso será una venta local, activando el componente `Lista`, el cual muestra la lista de productos que se encuentran disponibles para una venta.

   Para que se realice el proceso anterior, el componente `ListaCom` hace una llamada al componente `Lista`, para que éste haga una consulta a la base de datos, afín de ver los productos que hay en existencia. A su vez éste retorna la lista de productos existentes.

   Posteriormente se elige el producto a vender y éste se muestra en el componente `Tabla`, que contiene todos los productos referentes a esa venta específica. Este componente interactúa con el componente `Consulta`, para calcular el costo de la venta mediante una consulta a la base de datos sobre el costo unitario del producto a vender, mostrando finalmente el costo calculado del pago total del producto(s) a vender.

2. **Quitar producto:** cuando se requiere eliminar un producto de la lista de venta, el usuario elige un elemento del componente `Tabla` eliminándolo de la lista. Este componente se encarga de recalcular el costo total (si es que existen más productos) de la venta.

3. **Cancelar venta:** cuando se desea cancelar una venta, se interactúa con el componente `Consulta`, indicando que se elimine lo referente a esa venta específica.

4. **Terminar venta:** cuando finalmente el cliente ha decido terminar su compra, se le indica al sistema que la transacción ha terminado y que deberá insertar en la base de datos, el costo de la venta, la cantidad y tipo de productos que han sido vendidos mediante la interacción con el componente `Insertar`.

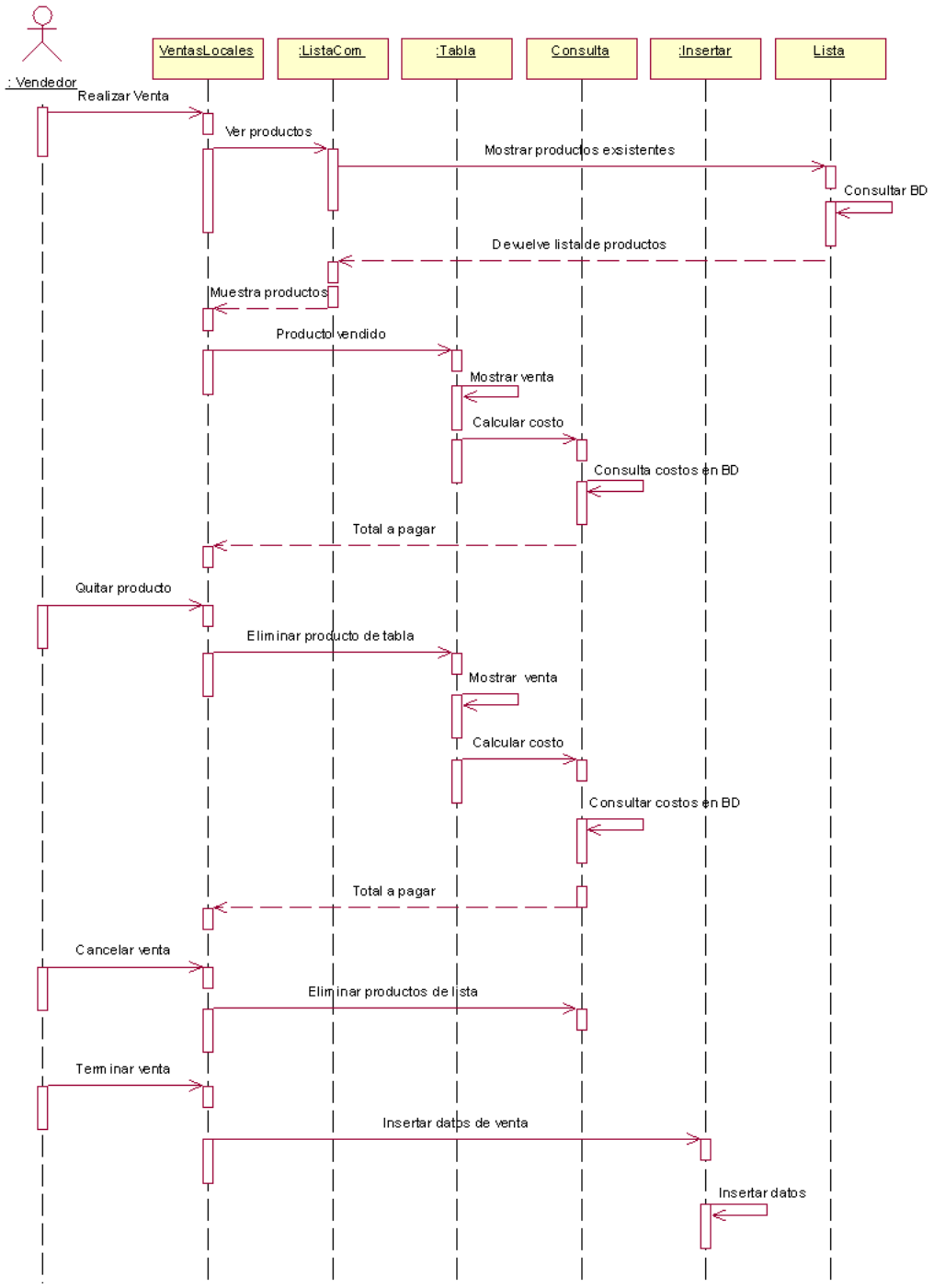

Figura 4.10: Diagrama de secuencia de componentes en ventas Locales

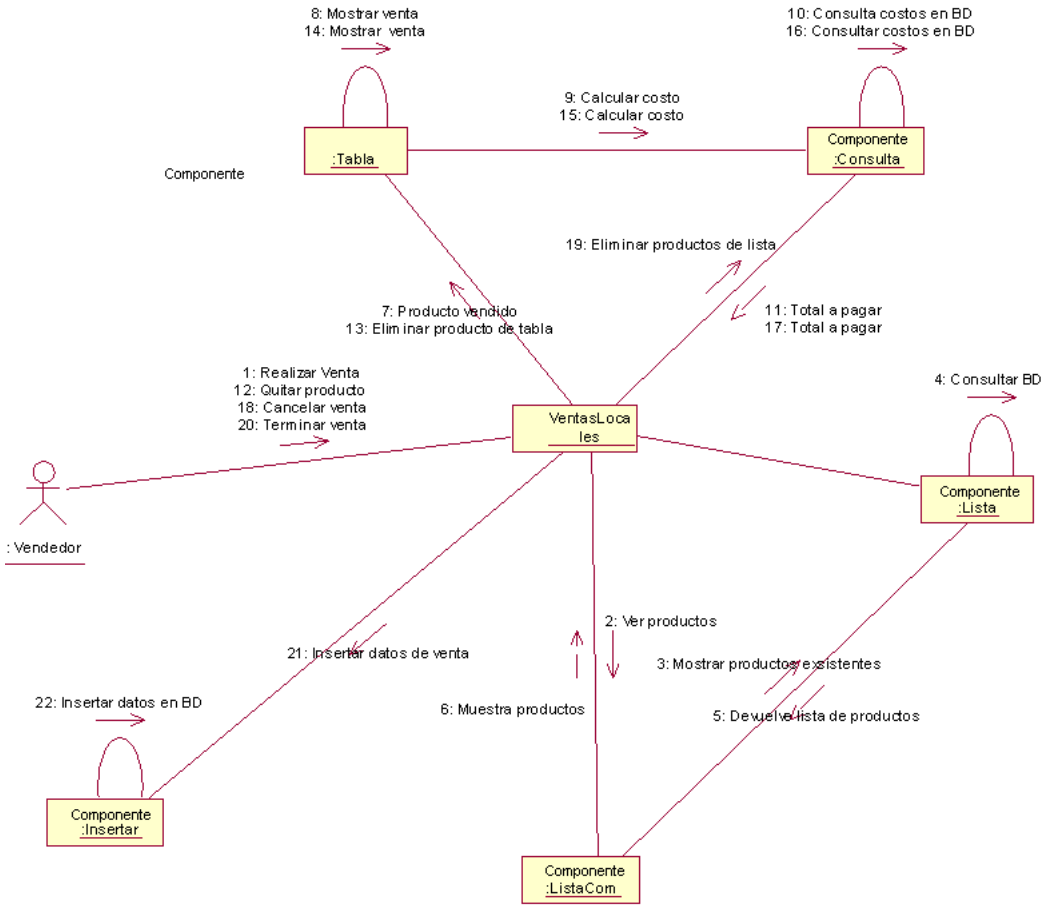

Figura 4.11: Diagrama de colaboración de componentes en ventas Locales

4.20.2 Diagrama de colaboración y secuencia - ventas telefónicas

En el diagrama de la figura 4.12 se muestra como el vendedor interactúa con el sistema para realizar una venta telefónica. Según la tarea que desee realizar se activa una serie de componentes para satisfacer la tarea requerida.

1. **Realizar venta:** cuando se requiere realizar una venta, el vendedor tiene que decir al sistema que tipo de venta debe realizar. En éste caso será una venta telefónica, así que se interactúa con el componente `Lista`, el cual muestra una lista de los clientes existentes, se elige un cliente y posteriormente se muestra la lista de productos existentes.

Para realizar lo anterior, `ListaCom` tiene que interactuar con el componente `Lista`, el cual realiza una consulta a la base de datos para mostrar la lista de los clientes que previamente han sido dados de alta. Una vez hecho ésto, se selecciona un cliente y comienza el proceso de venta.

Cuando el proceso de venta ha iniciado se activa el componente `Lista`, el cual le muestra la lista de productos que se encuentran disponibles para una venta.

Para que se realice el proceso anterior, el componente `ListaCom` hace una llamada al componente `Lista`, para que éste haga una consulta a la base de datos, para ver los productos que hay en existencia, a su vez retorna la lista de productos existentes.

Posteriormente se elige el producto a vender y éste se muestra en una tabla que contiene todos los productos referentes a esa venta específica, mediante el componente `Tabla`.

El componente `Tabla` tiene que interactuar con el componente `Consulta` para calcular el costo de la venta mediante una consulta a la base de datos sobre el costo unitario del producto a vender, mostrando finalmente el costo calculado del pago total del producto(s) a vender.

2. **Quitar producto:** cuando se requiere eliminar un producto de la lista de ventas, el usuario elige un elemento del componente `Tabla` eliminándolo de la lista. Este componente se encarga de recalcular el costo total (si es que existen más productos) de la venta.

3. **Cancelar venta:** cuando se desea cancelar una venta, se interactúa con el componente `Consulta`, indicando que se elimine lo referente a esa venta específica.

4. **Terminar venta:** cuando finalmente el cliente ha decido terminar su compra, se le indica al sistema que la transacción ha terminado y que deberá insertar en la base de datos, el costo de la venta, la cantidad y tipo de productos que han sido vendidos mediante la interacción con el componente `Insertar`.

5. **Alta de cliente:** cuando se comienza el proceso de ventas telefónicas y en el listado de clientes existentes no aparece el cliente al cual se está atendiendo, éste tiene que ser dado de alta en el sistema. Para ello a través del componente `Insertar`, se registran los datos generales del cliente en la base de datos.

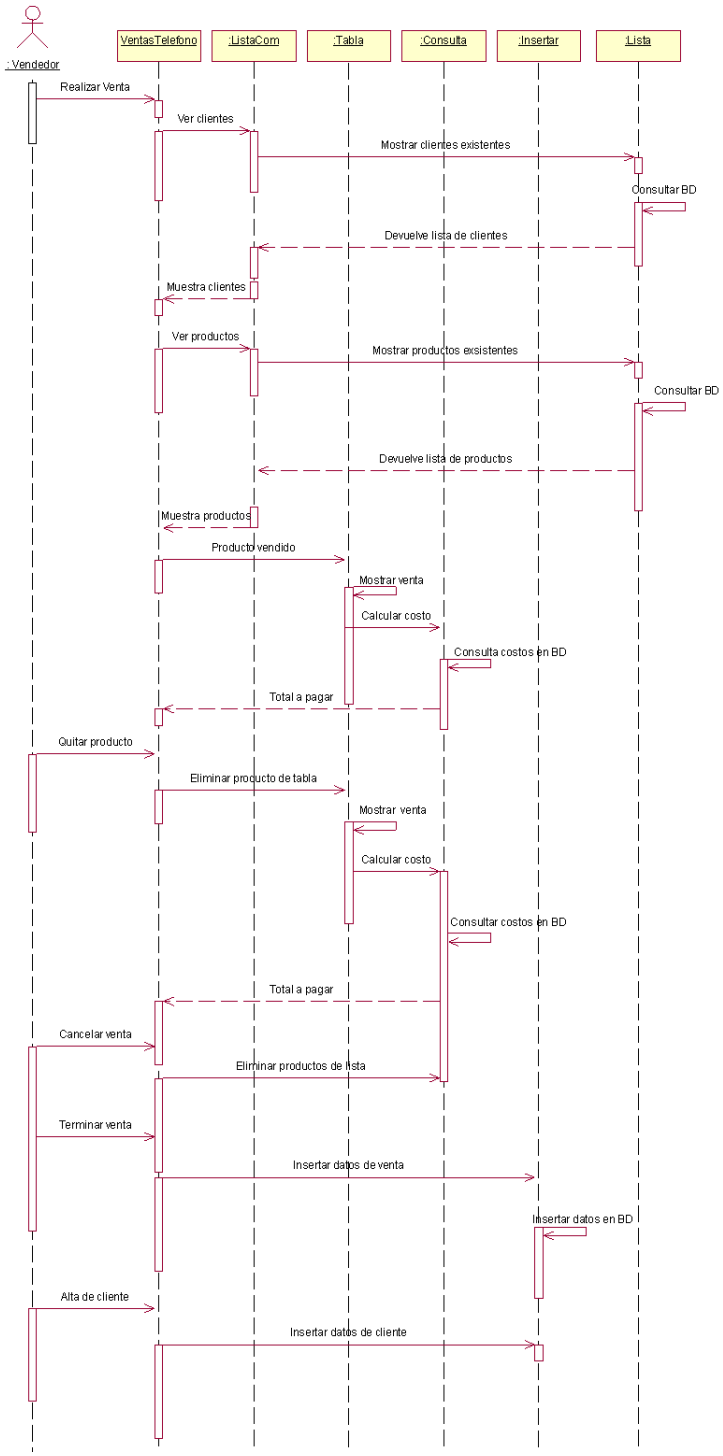

Figura 4.12: Diagrama de secuencia de componentes en ventas telefónicas

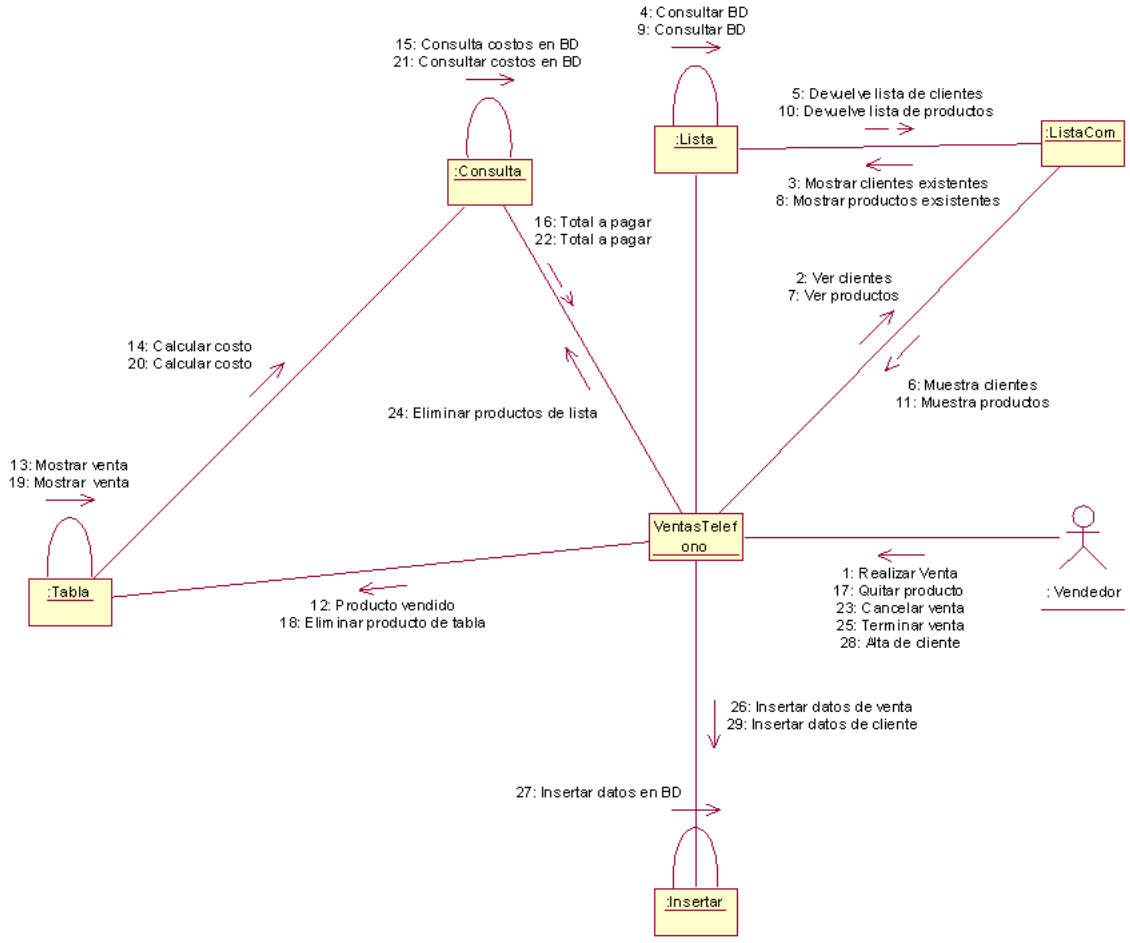

Figura 4.13: Diagrama de colaboración de componentes en ventas telefónicas

4.20.3 Diagrama de colaboración y secuencia - administrador

Este diagrama muestra (ver figura 4.14) la interacción que el administrador del sistema tiene con la aplicación. Según la tarea que desee realizar se activa el o los componentes que realizan la tarea.

1. **Alta de datos:** cuando el administrador desea realizar cualquier tipo de alta, por medio de la interfaz del sistema, se activa el componente `Insertar`, el cual registra los datos que el administrador ha capturado mediante el sistema.

2. **Modificar datos:** en el momento en el que el administrador desea realizar una modificación, interactúa con el componente `ListaCom` que invoca a `Lista` la cual hace una consulta a la base de datos para obtener un conjunto de datos, de entre éstos datos el usuario elige los registros que desea para su modificación.

Una vez hecha la operación, el usuario realiza la modificación deseada y por medio del componente `Insertar`, registra la modificación realizada.

3. **Eliminar datos:** el usuario selecciona de una lista de registros, que es proporcionada por el componente `Lista`, el dato a eliminar y lo borra del sistema mediante el componente `Insetar`.

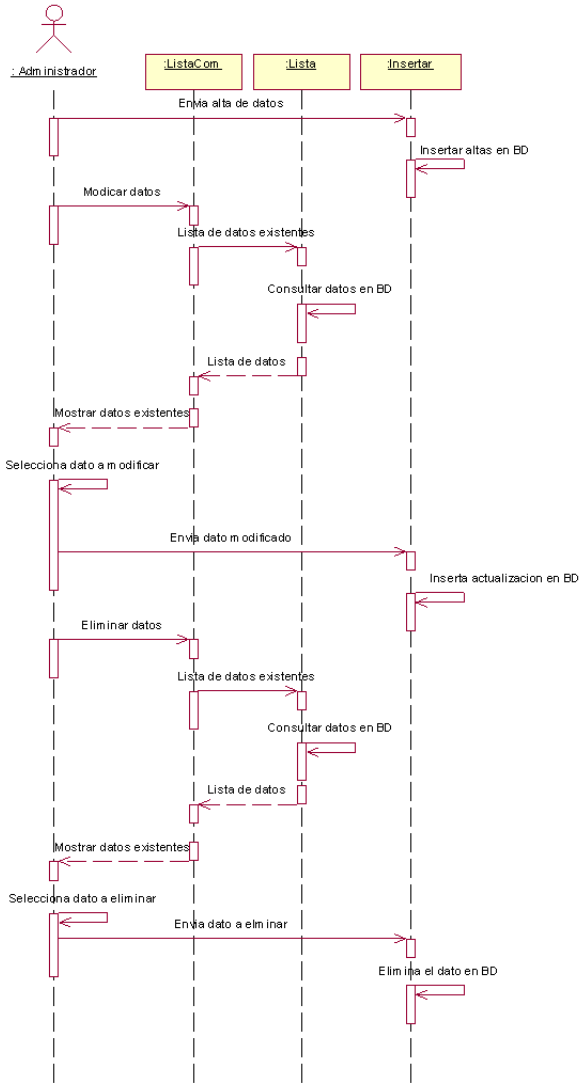

Figura 4.14: Diagrama de secuencia de componentes para el administrador

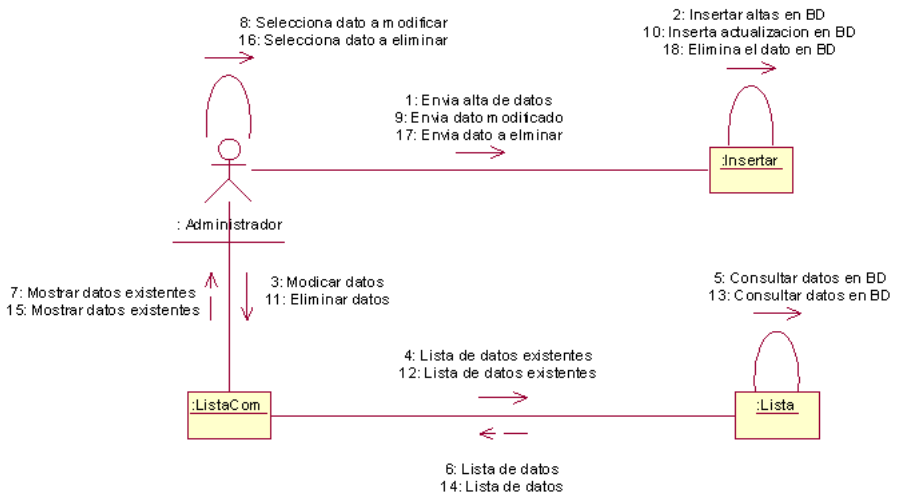

Figura 4.15: Diagrama de colaboración de componentes para el administrador

4.21 Discusión sobre el desarrollo del sistema

Establecer un comparativo para decidir que *middleware* es el mejor, es un poco complicado debido a que influyen varios factores, tales como: la experiencia del desarrollador, el conocimiento del lenguaje de programación utilizado para el desarrollo del sistema y principalmente el conocimiento de la herramienta *middleware* a utilizar.

En nuestro caso se desarrollo un sistema de ventas. El sistema se implemento de la misma forma y con las mismas características en cada una de las herramientas *middleware* estudiadas: EJB, DCOM y CCM.

Una de las características del sistema es que se debería ejecutar en un ambiente Web, para lo cual EJB y DCOM no tienen mayor problema debido a que estas herramientas utilizan JSP´s y ASP´s respectivamente, sin embargo, en el caso de CCM fue un poco más elaborado el ejercicio, ya que se tenía que buscar la manera en que los componentes CORBA interactuaran con el entorno Web.

El problema de la navegación en internet con CORBA fue resuelto mediante el uso del protocolo IIOP de CORBA, el cual permite enviar y recibir mensajes vía TCP/IP, y la implementación de un módulo JavaBean que comunica la aplicación con un servidor J2EE.

Para el caso del desarrollo, se programaron los mismos componentes siguiendo la misma lógica en todos los casos con las mismas herramientas, debido a que el uso

de componente hace que una aplicación se vuelva transparente, solo fue necesario programar cada componente en conjunto con sus interfaces y de esta manera usarlo como una caja negra. Cada componente se encarga de recibir información y de dar un resultado, bajo esta tendencia es como se integran dentro del sistema.

Por lo tanto, hemos concluido que la utilización de una herramienta determinada solo depende del desarrollador, quien podrá basarse en las características que hemos mencionado en el presente capítulo y que se encuentran plasmadas en la tabla 4.1. Esta tabla le proporcionará una visión de las capacidades de cada sistema para así poder tomar una decisión en base a sus necesidades.

De nuestro análisis hemos obtenido que una determinada herramienta se puede utilizar, en los casos previstos en las secciones: 4.21.1, 4.21.2 y 4.21.3.

4.21.1 DCOM

- Es necesario elaborar un desarrollo rápido.
- La seguridad no es muy necesaria.
- La necesidad de persistencia de información es nula o casi nula.
- Se requiere utilizar de componentes desarrollados en diferentes lenguajes de Microsoft, en cuyo caso se debe utilizar Windows y Visual Studio.
- Solo se cuenta con plataformas Windows.
- La lógica del negocio lo permite.

4.21.2 J2EE-EJB

- Se requieren aplicaciones robustas.
- Es necesaria la interacción con diferentes plataformas de Sistemas Operativos.
- Se requiere la implementación de seguridad.
- Se requiere la implementación de persistencia.
- Se quiere implementar una aplicación multinivel.
- Se requiere utilizar en gran medida el lenguaje Java dentro del desarrollo.

4.21.3 CORBA-CCM

- Se requieren aplicaciones muy robustas.
- Es necesaria la interacción con diferentes plataformas de Sistemas Operativos.
- Se requiere la implementación de un alto grado de seguridad.
- Se requiere la implementación de un alto grado de persistencia.
- Se quiere implementar una aplicación multinivel.
- Se tienen sistemas heredados en diferentes lenguajes soportados por CORBA y es necesario interactuar con ellos.

Capítulo 5

Conclusión y trabajo futuro

El estudio que realizamos, ha cubierto un número significativo de características, mismas que fueron necesarias para comprender cual es el fondo y la forma de nuestra discusión.

Estudiamos el paradigma de componentes y las diferencias que éste enfrenta con respecto al paradigma de objetos. Se dieron algunas definiciones y formamos una que a nuestro parecer es la más adecuada, así como también se revisó la ingeniería basada en componentes, misma que es necesaria para realizar cualquier desarrollo bajo este paradigma.

Se definió que es una herramienta *middleware* y cuales son las características que ésta debe de tener, así como los servicios que presta y las ventajas que ofrece para el desarrollo de los sistemas distribuidos.

Hemos examinado las características de CORBA-CCM, DCOM y J2EE-EJB, en torno a dos vertientes. Por un lado se describen las características generales de cada *middleware* y por el otro se discute el desarrollo de aplicaciones distribuidas con componentes.

Recordemos que el paradigma orientado a componentes no es más que una extensión del paradigma orientado a objetos, es decir que el modelo de objetos es la base del modelo de componentes. Es por ello que en ocasiones los desarrolladores no visualizan la diferencia entre un modelo y otro.

La principal diferencia en un paradigma y otro radica en el nivel de encapsulación, reutilización, acoplamiento e implementación que se llega a lograr mediante el uso de componentes, así como la desaparición de la necesidad de conocer la estructura de un componente para ser implementado, pues siempre se piensa en el concepto de caja negra.

Para el desarrollo de aplicaciones, es siempre posible utilizar el modelado orientado a objetos, no perdiendo de vista que en vez de utilizar clases se utilizan componentes.

Sin embargo, antes de comenzar a modelar en forma, se tienen que identificar los componentes que intervendrán en el sistema. Para ello hay que utilizar una técnica de la programación estructurada, conocida como descomposición funcional, mediante esta técnica se descubre la colección de componentes a utilizar en la aplicación.

Después de haber obtenido la colección de componentes, es posible ahora sí, comenzar a modelar la arquitectura de la aplicación, en donde es posible incluir varios tipos de componentes, los cuales pueden ser propios o bien adquiridos mediante un tercero, en ésta arquitectura, se establecen las partes del sistema en donde existirá o no, composición de componentes (es decir trabajo coordinado).

Desafortunadamente, aun en la actualidad (que no debiera de ser así), existen algunos problemas con respecto al paradigma de componentes, pues hay componentes que no son compatibles con otros – tal es el caso de Java, en donde solo puede utilizar componentes fabricados para este lenguaje -, para lo cual es necesaria la utilización de interfaces y/o adaptadores que permitan interactuar con éstos y con un lenguaje específico.

Conclusión

Después de un exhaustivo análisis de las arquitecturas más utilizadas en la actualidad para el desarrollo de sistemas distribuidos y de la forma que implementan el paradigma de componentes, podemos concluir, en base a los resultados obtenidos que la herramienta más completa para el desarrollo de aplicaciones basadas en componentes es CORBA-CCM, seguida de J2EE-EJB.

Aunque J2EE podría verse limitada por el soporte del lenguaje, en donde Java es el único soportado por los componentes, en contraste con DCOM, que es multilenguaje. La potencialidad que ofrece Java, mediante las implementaciones existentes de un gran número de servicios ofrecidos para ser utilizados, es lo que hace a esta herramienta tan robusta, dejando en último lugar a DCOM, que solo soporta las implementaciones relacionadas con Windows.

Hemos llegado a la conclusión que el uso de un *middleware* específico para el desarrollo de una aplicación distribuida con componentes, siempre dependerá del dominio del sistema y de las necesidades que se requieran cubrir. Por lo anterior, solo proponemos los siguientes puntos para la utilización de una determinada herramienta:

- Cuando se requiera desarrollar una aplicación en donde la estructura, funcionalidad total de la aplicación así como la lógica del negocio, necesiten ser definidas por el desarrollador, se propone la utilización de CORBA-CCM.

- Cuando el sistema a desarrollar requiere enfocarse exclusivamente en la lógica del negocio y cuando que se tenga que dejar de lado, o simplemente no se desee configurar o en su caso programar alguna funcionalidad, como por ejemplo la conexión a bases de datos, se podrá utilizar J2EE-EJB.

- Si lo que se requiere es un desarrollo rápido, en donde la lógica del negocio es lo que más importa, se recomienda utilizar DCOM, con la reserva de que se deberá tener una plataforma Windows, en otro caso se podrá utilizar J2EE-EJB.

- CORBA-CCM es robusto en cuanto a los servicios que brinda. La descomposición funcional genera como resultado un conjunto de componentes que podrán ser desarrollados a gusto del desarrollador de la aplicación, sin ninguna limitación y que es una de las ventajas que ofrece CORBA.

Como punto final, solo resta añadir que no pretendemos decir, qué herramienta es mejor o peor, el desarrollo y el *middleware* a utilizar siempre dependeran de los desarrolladores, y de las caracteristicas del sistema a desarrollar. Solo hemos dado un conjunto de parámetros que pueden ayudar fácilmente a elegir la herramienta que más se adecué a las necesidades de los desarrolladores.

Trabajo futuro

De acuerdo a nuestro estudio, hemos encontrado los siguientes problemas, que hemos dejado como trabajo futuro de investigación:

1. El estudio de los modelos orientados a componentes proporciona una gran capacidad para el desarrollo de sistemas, sin embargo hay que evaluar el nivel de seguridad que cada herramienta *middleware* proporciona en el desarrollo de sistemas distribuidos. Así debemos establecer un análisis cuantitativo que permita estimar el grado de seguridad proporcionado por cada *middleware*, y en qué casos se debe utilizar para realizar el desarrollo de determinado sistema.

2. Es posible establecer un estudio comparativo de las diferentes implementaciones para CCM, en cuanto a calidad de la implementación, soporte de lenguajes de programación e interoperabilidad con otras implementaciones para CORBA-CCM y el soporte de intercambio de información con otros modelos de componentes como DCOM y J2EE.

3. Establecer un estudio comparativo entre los diferentes *middleware* orientados a componentes con respecto al grado de interoperabilidad que soportan con respecto a otros *middleware*, tanto orientados a componentes como orientados a objetos.

Apéndice A

Código fuente de los componentes de la aplicación

A.1 Componentes DCOM

A.1.1 Componente para inserción

```
1  Option Explicit
2  'Un componente que nos permite administra las
3  'insercciones dentro de la base de datos.
4
5  Public Sub insertaDatos(cadena As Variant)
6
7
8          ' variable que se encargara
9      ' de la base de datos
10         Dim BDD As ADODB.Connection
11
12         ' variable que se encarga de
13     'recorrer la tabla en la BD
14         Dim TABLA As ADODB.Recordset
15
16         'variable que contendra la consulta SQL
17         Dim SQL As String
18
19         Dim COMANDOS As ADODB.Command
20
21         Set BDD = New ADODB.Connection
22         Set COMANDOS = New ADODB.Command
23 'BDD.Open permite crear el acceso
24 'a la base de datos, así como la
25 'autentificación del usuario que
26 'la ha de utilizar.
```

```
27
28 BDD.Open "Driver={PostgreSQL␣UNICODE};
29 Server=192.168.1.101;
30 Port=5432;Database=comidas;
31 Uid=postgres;
32 Pwd=postgres;"
33
34    'variables que gestionan la consulta a la BD
35         SQL = cadena
36         Set TABLA = New ADODB.Recordset
37
38 'Apartir del Witch se integra la sentencia
39 'SQL y la cadena que sea de insertar
40 'se establece la conexion a la base
41 'de datos y se ejecuta la inserción.
42         With COMANDOS
43         .CommandText = SQL
44         .CommandType = adCmdText
45         .ActiveConnection = BDD
46         .Execute
47         End With
48
49 'Liberamos el componente
50 'de la memoria.
51
52 Set BDD = Nothing
53 Set TABLA = Nothing
54 End Sub
```

A.1.2 Componente para consulta y conexión a la base de datos

```
1  Option Explicit
2  'El siguiente componente se encarga de la administración
3  'de la conexión a la base de datos, así como la gestión
4  'de consultas y retorna el resultado de la misma.
5
6  Public Function listar(Cadena As Variant) As Variant
7
8     'variables para gestion de la conexión a la BD
9          Dim objCnx As ADODB.Connection
10         Dim objCmd As ADODB.Command
11         Dim objRst As ADODB.Recordset
12
13     'variable que gestiona la consulta del cliente
14         Dim strCnx As String
```

```
15
16 'Gestion de la conexión a la base de datos
17 'mediante el ADODB y su Driver correspondiente.
18         strCnx = "Driver={PostgreSQL␣UNICODE};
19 Server=192.168.1.101;
20 Port=5432;
21 Database=comidas;
22 Uid=postgres;
23 Pwd=postgres;"
24
25     'conexión y realización de la consulta
26         Set objCnx = New ADODB.Connection
27         objCnx.Open strCnx
28         Set objCmd = New ADODB.Command
29         objCmd.ActiveConnection = objCnx
30         objCmd.CommandText = Cadena
31         Set objRst = New ADODB.Recordset
32
33    'almacenamiento del resultado en un vector
34         objRst.Open objCmd, , , , adCmdText
35
36     'retorno del resultado dentro de un vector
37         listar = objRst.GetRows
38
39 End Function
```

A.1.3 Componente para despliegue de datos

```
1  Option Explicit
2  'El siguiente componente toma el resultado del
3  'componente Listar y genera una lista desplegable
4  'en formato html cuyo contenido es el resultado
5  'de la consulta realizada.
6
7  Public Sub generaTabla(ByVal Cadena As Variant)
8          Dim objContext As ObjectContext
9          Dim objResponse As Response
10         Dim objCom As lista.List
11         Dim rec()
12         Dim strHTML As String
13         Dim j, i As Integer
14
15         Set objContext = GetObjectContext
16         Set objResponse = objContext("Response")
17         Set objCom = New lista.List
18
19         rec = objCom.listar(Cadena)
```

```
20      'se da formato al resultado generado
21      ' para ser visto en
22      'la pagina html
23
24      'se recorre el vector que contiene el
25      'resultado de la consulta hasta llegar
26      'a EOF
27          For i = 0 To UBound(rec, 2)
28            For j = 1 To UBound(rec, 1)
29              strHTML = strHTML & "<option␣value="
30                & rec(0, i) & ">" &
31                   rec(1, i) & "</option>"
32            Next
33          Next
34      'escribe el resultado como salida html
35      'para ser gestionado por una página
36      'ASP
37          objResponse.Write strHTML
38  End Sub
```

A.2 Componentes EJB

A.2.1 Componente Home

```
1  //ConectarHome.java
2  //La clase Home crea la interfaz que
3  //permite crear las instancias
4  //del bean al cliente
5  package componente;
6  import java.rmi.RemoteException;
7  import javax.ejb.CreateException;
8  import javax.ejb.EJBHome;
9
10 public interface ConectarHome extends EJBHome{
11      Conectar create() throws RemoteException, CreateException;
12 }
```

A.2.2 Componente Remote

```
1  //Conectar.java
2  /*Esta clase crea la interfaz remota en la
3  que se declaran los métodos que el bean
4  podrá instanciar para realizar determinadas
5  operaciones.
6  */
7  package componente;
```

```
 8 import javax.ejb.EJBObject;
 9 import java.rmi.RemoteException;
10 import java.sql.*;
11 import java.io.*;
12
13
14 public interface Conectar extends EJBObject{
15     public String GetConsulta()
16         throws RemoteException;
17     public void SetConsulta(String _consulta)
18         throws RemoteException;
19     public String GetListar()
20         throws RemoteException;
21     public void SetListar(String cadena)
22         throws RemoteException;
23     public void SetTabla(String cadena)
24         throws RemoteException;
25     public void SetConsultaSimple(String cadena)
26         throws RemoteException;
27     public void SetModificacion(String cadena)
28         throws RemoteException;
29     public void SetDesconectar()
30         throws RemoteException;
31 }
```

A.2.3 Componente BeanListar

```
1  //BeanListar.java
2  /* La clase BeanListar, es el componente
3  que implementa todos los métodos especificados
4  por la interfaz remota y los especificados
5  por la interfaz home.*/
6
7  package componente;
8  import java.rmi.RemoteException;
9  import javax.ejb.SessionBean;
10 import javax.ejb.SessionContext;
11 import java.sql.*;
12
13 public class BeanListar implements SessionBean {
14
15      /*Declaramos las variables que servirán
16      para la gestión de la conexion y
17      autenticación a la base de datos,
18      administrada por el pool de conexión
19      ConnectionPool.*/
20
```

```
21      String driver = "org.postgresql.Driver";
22      String url =
23      "jdbc:postgresql://148.247.102.23:5432/comidas?+
24 user=postgres&password=postgres";
25
26      String consulta ="";
27      String resultado="";
28      ConnectionPool pool;
29      Connection connection;
30      Statement stmt;
31      ResultSet rs;
32
33     /*Mediante el contructor de BeanListar
34     se crea una instancia de ConnectionPool
35     y se inicializa para realizar la
36     conección a la base de datos.*/
37     public BeanListar(){
38        try{
39             pool = new ConnectionPool();
40             pool.Conectar(driver,url,
41                           "postgres",
42                           "postgres",
43                            10,50,true);
44             connection = pool.getConnection();
45            /*Si la conección falla se crea
46            una excepción, informando cual
47            fue el problema*/
48            }catch(SQLException e){
49          System.out.println("Erro:"+e);
50       }
51    }
52      /*Retorna el resultado de la consulta
53      realizada*/
54      public String GetConsulta(){
55          return (consulta);
56      }
57
58      /*Obtiene la cadena que tiene el tipo
59      de consulta SQL que se desea realizar
60      y ésta se pasa como parametro a
61      SetListar*/
62      public void SetConsulta(String _consulta){
63           consulta = _consulta;
64           SetListar(consulta);
65      }
66
```

```
67      /*Retorna el resultado generado por
68      SetListar*/
69       public String GetListar (){
70           return resultado;
71       }
72
73       /*Se obtiene una cadena con formatado de
74       lista desplegable en html como resultado
75       de la consulta realizada*/
76
77       public void SetListar(String cadena){
78            /*se establece la conexión a la base de datos*/
79            ConnectionPool pool = new ConnectionPool();
80            /*se realiza la consulta a la base de datos*/
81            try{
82                Statement stmt = connection.createStatement();
83                ResultSet rs = stmt.executeQuery(cadena);
84                resultado="";
85
86                /*Se recorre rs hasta llegar a EOF y al mismo
87                tiempo se almacena en resultado la cadena
88                resultante con formato de html*/
89                while (rs.next()){
90                    resultado =resultado+ "<option␣value="+
91                                rs.getString(1)+">"+
92                                rs.getString(2)+
93                                "</option>"+"\n";
94                }
95           /*Si ocurre un error se crea una excepción
96           para que el sistema siga su funcionamiento
97           normal y se informa del tipo de error
98           generado*/
99           }catch (SQLException e){
100               System.err.println("Error␣al␣general"+
101                                  "la␣asociación:" + e);
102               /*se cierra la conexión realizada*/
103               pool = null;
104          }
105      }//fin del método SetListar
106
107       /*Se obtiene una cadena con formatado de
108      tabla en html como resultado
109      de la consulta realizada*/
110
111       public void SetTabla(String cadena){
112            /*se establece la conexión a la base de datos*/
```

```
113           ConnectionPool pool = new ConnectionPool();
114
115           try{
116
117              /*se realiza la consulta a la base de datos*/
118               Statement stmt = connection.createStatement();
119               ResultSet rs = stmt.executeQuery(cadena);
120
121               ResultSetMetaData metaData = rs.getMetaData();
122               int count = metaData.getColumnCount();
123
124               resultado="";
125               resultado="<table␣border=1>"+"\n";
126
127                 /*Se recorre rs hasta llegar a EOF y al mismo
128               tiempo se almacena en resultado la cadena
129               resultante con formato de html*/
130               while (rs.next()){
131                  resultado+="<tr>"+"\n";
132                  for (int i = 1; i<=count; i++){
133                     resultado +="<td>"+
134                       rs.getString(metaData.getColumnName(i))+
135                               "</td>"+"\n";
136                  }
137                  resultado = resultado + "</tr>";
138               }
139               resultado = resultado + "</table>";
140
141          /*Si ocurre un error se crea una excepción
142           para que el sistema siga su funcionamiento
143           normal y se informa del tipo de error
144           generado*/
145           }catch (SQLException e){
146                System.err.println("Error␣al␣general"+
147                 "la␣asociación:" + e);
148
149                /*se cierra la conexión realizada*/
150                pool = null;
151           }
152       }//fin del metodo SetTabla
153
154
155        /*Este método unicamente obiene un registro como
156        resultado de la consulta realizada*/
157
158        public void SetConsultaSimple(String cadena){
```

```
159           ConnectionPool pool = new ConnectionPool();
160
161           try{
162               Statement stmt = connection.createStatement();
163               ResultSet rs = stmt.executeQuery(cadena);
164               ResultSetMetaData metaData = rs.getMetaData();
165               int count = metaData.getColumnCount();
166
167               resultado="";
168
169               /*Se recorre rs hasta llegar a EOF */
170
171               while (rs.next()){
172
173                  for (int i = 1; i<=count; i++){
174                   resultado +=
175                   rs.getString(metaData.getColumnName(i));
176                  }
177               }
178
179              /*Si ocurre un error se crea una excepción
180           para que el sistema siga su funcionamiento
181           normal y se informa del tipo de error
182           generado*/
183
184           }catch (SQLException e){
185                System.err.println("Error al general"+
186                     " la asociación:" + e);
187                pool = null;
188           }
189       }//fin del metodo Consulta Simple
190
191
192       /*Este método permite realizar modificaciones
193       a los registro de la base de datos, segun
194       la sentencía SQL pasada como argumento del
195       método*/
196       public void SetModificacion(String cadena){
197           ConnectionPool pool = new ConnectionPool();
198
199            try{
200                Statement stmt = connection.createStatement();
201                stmt.execute(cadena);
202                //pool.free(connection);
203            }catch (SQLException e){
204                System.err.println("Error al general"+
```

```
205                 "la asociación:" + e);
206               pool = null;
207          }
208
209
210      }//fin del metodo SetModificacion
211
212      //Libera las conexiones que se hayan realizado
213       public void SetDesconectar(){
214            pool.free(connection);
215     }//fin de SetDesconectar
216
217      public void ejbCreate() {}
218      public void ejbRemove() {}
219      public void ejbActivate() {}
220      public void ejbPassivate() {}
221      public void setSessionContext(SessionContext sc) {}
222 }\\
```

A.2.4 Pool de conexiones para Java

```
1  package componente;
2  import java.sql.*;
3  import java.util.*;
4  import java.net.*;
5  import javax.sql.DataSource.*;
6  import java.io.*;
7  import javax.naming.*;
8
9  // Una clase para asignar previamente, reciclar y administrar
10 // las conexiones JDBC
11
12
13 public class ConnectionPool implements Runnable {
14       private String driver, url, username, password;
15       private int maxConnections;
16       private boolean waitIfBusy;
17       private Vector availableConnections, busyConnections;
18       private boolean connectionPending = false;
19
20       public ConnectionPool(){
21
22       }//fin del constructor
23
24
25       //Realiza las conexiones para la base de datos
26       public void Conectar(String driver, String url,
```

```
27                              String username, String password,
28                              int initialConnections,
29                              int maxConnections,
30                              boolean waitIfBusy)
31              throws SQLException {
32
33          this.driver = driver;
34          this.url = url;
35          this.username = username;
36          this.password = password;
37          this.maxConnections = maxConnections;
38          this.waitIfBusy = waitIfBusy;
39          if (initialConnections > maxConnections) {
40             initialConnections = maxConnections;
41          }
42          availableConnections = new Vector(initialConnections);
43          busyConnections = new Vector();
44          for(int i=0; i<initialConnections; i++){
45              availableConnections.addElement(makeNewConnection());
46          }
47       }//fin de constructor
48
49       public synchronized Connection getConnection()
50             throws SQLException {
51          if (!availableConnections.isEmpty()){
52             Connection existingConnection =
53                 (Connection)availableConnections.lastElement();
54             int lastIndex = availableConnections.size() - 1;
55             availableConnections.removeElementAt(lastIndex);
56             //Si la conexión en la lista disponible
57             //está cerrada (por exceder el tiempo
58             //de espera), entonces quitar de la lista
59             //de disponibles y repetir el proceso
60             //para obtener una conexión. También
61             //indíquelo a los subprocesos que
62             //aguardaban una conexión dado que se
63             //llego al límite de maxConnection
64
65             if (existingConnection.isClosed()) {
66                //Se ha liberado un lugar para cualquiera que
67                //espere
68                notifyAll();
69                return(getConnection());
70             }
71             else {
72                  busyConnections.addElement(existingConnection);
```

```
73                  return(existingConnection);
74              }
75          }
76          else {
77              //Tres casos posibles:
78              //1) No ha llegado al límite de maxConnection.
79              //Por ello, establecer una en segundo plano
80              //si no hay una pendiente y luego aguarde la
81              //siguiente conexión disponible( que no
82              //necesariamente debe ser la recién establecida).
83              //2) Ha llegado al límite de maxConnections y
84              //el indicador waitIfBusy tiene un valor de false.
85              //Arroje la SQLException en tal caso.
86              //3) Ha llegado al límite maxConnections y el
87              //indicador waitBusy tiene un valor de true.
88              //Entonces, haga lo mismo que en la segunda
89              //parte del paso 1: aguarde la disponibilidad
90              //de una conexión.
91
92              if ((totalConnections() < maxConnections) &&
93                  !connectionPending) {
94                  makeBackgroundConnection();
95              }
96               else if (!waitIfBusy) {
97                   throw new SQLException("Se ha llegado al" +
98                                          "limite de conexiones");
99               }
100             //Esperar ya sea a que se establezca
101             //una nueva conexión 8si ejecutó a
102             //makeBackgroundConnection) o que una
103             //conexión existente sea liberada.
104             try {
105                 wait();
106             } catch (InterruptedException ie) { }
107             //Alguien ha liberado una conexión,
108             //intente nuevamente.
109             return (getConnection());
110         }
111     }//fin de getConnection
112
113     //No podrá tan sólo hacer una nueva conexión en
114     //primer plano cuando no haya ninguna disponible,
115     //dado que podrá tomar varios segundos con una
116     //conexión de red lenta. En vez de ello, aranque
117     //un subproceso que establezca una nueva conexión,
118     //y guarde. Se le avisará ya sea cuando cuando se
```

```
119         //establezca una nueva conexión o si alguien deja
120         //de usar una existente.
121
122         private void makeBackgroundConnection(){
123             connectionPending = true;
124             try {
125                 Thread connectThread = new Thread(this);
126                 connectThread.start();
127             } catch (OutOfMemoryError oome){
128               //Renuncie a la nueva conexión
129             }
130         }//fin de makeBackgroundConnection
131
132         public void run(){
133             try{
134                 Connection connection = makeNewConnection();
135                 synchronized(this) {
136                    availableConnections.addElement(connection);
137                    connectionPending = false;
138                    notifyAll();
139                 }
140               } catch (Exception e){// SQLException u OutOfMemory
141                 // Reiniciar a la nueva conexión y aguarde que una
142                 // este disponible
143               }
144         }//fin de run()
145
146         //Esto hace explicitamente una nueva conexción.
147         //Se ejecuta en primer plano cuando se establecen
148         //los valores iniciales de ConnectionPool, y se
149         //ejecuta en segundo plano cuando está en ejecución.
150
151         private Connection makeNewConnection ()
152                throws SQLException {
153            try {
154               //Cargar el controlador de la BD
155               // si no esta cargado
156                Class.forName(driver);
157                //Establezca la conexión por la red
158                // a una base de datos
159                Connection connection =
160                    DriverManager.getConnection(
161                                 url, username, password);
162                    return(connection);
163             } catch (ClassNotFoundException cnfe) {
164                  throw new SQLException("No se encontro"+
```

```
165                 "la clase del"+"controlador:" + driver );
166             }
167         }// fin de makeNewConnection
168
169         public synchronized void free(Connection connection){
170               busyConnections.removeElement(connection);
171               availableConnections.addElement(connection);
172               //Indicar a los subprocesos que
173               // aguardan una conexión
174               notifyAll();
175         }//fin de free()
176
177         public synchronized int totalConnections(){
178              return(availableConnections.size() +
179                     busyConnections.size());
180         }//fin de totalConnections
181
182         /** Cierre todas las conexiones. Utilizar
183          *  con precuausión: asegúrese de hacer
184          *  notar que no es necesario ejecutar
185          *  este procedimiento cuando se termina
186          *  con una ConnectionPool, dado que se
187          *  garantiza que todas las conexiones se
188          *  cierran cuando se les recolecta la basura.
189          *  Pero este método otorga un mayor control
190          *  cuando se tienen que cerrar las conexiones.
191          */
192
193          public synchronized void closeAllConnection() {
194              closeConnections(availableConnections);
195              availableConnections = new Vector();
196              closeConnections(busyConnections);
197              busyConnections = new Vector();
198          }//fin de closeAll()
199
200          private void closeConnections(Vector connections) {
201           try {
202              for ( int i = 0; i<connections.size(); i++){
203                  Connection connection =
204                     (Connection)connections.elementAt(i);
205                   if(!connection.isClosed()){
206                      connection.close();
207                   }
208               }
209            } catch (SQLException sqle){
210              // Ignorar los errores; de todas formas recolectar la
```

```
211             // baseura
212           }
213         }// fin de closeConnections()
214
215         public synchronized String toString() {
216              String info =
217                "\n␣ConnectionPool1(" + url + "," + username + ")"+
218                ",\n␣disponibles=" + availableConnections.size() +
219                ",\n␣ocupadas␣=" + busyConnections.size() +
220                ",\n␣máximas=" + maxConnections;
221               return(info);
222         }//fin de toString()
223
224 }//fin de la clase ConnectionPool
```

A.3 Componentes CCM

A.3.1 Archivo IDL

```
1  //ccm.idl
2  /*Creamos la interfaz del componente
3  mediante lenguaje IDL*/
4
5  interface CCM_SQL {
6    /*El componente contendra
7    un método para gestionar
8    las consultas*/
9    void enviarConsulta ();
10 };
11
12 /*Creamos el componente que
13 dara soporte al sistema*/
14
15 component Consulta_CCM supports CCM_SQL {
16   /*El siguiente atributo permite
17   que se gestionene los mensajes
18   enviados por los clientes al
19   componente*/
20   attribute string message;
21 };
22
23 //Home crea la interfaz que
24 //permite crear las instancias
25 //del componente CCM al cliente
26
27 home CCMHome manages Consulta_CCM {
```

```
28
29   attribute string initial_message;
30 };
```

```
1  //cadsql.idl
2  /*La siguiente interfaz declara
3  al cliente CORBA que se comunica
4  con el componente CCM_SQL
5
6  /*Esta interfaz contiene al método
7  que obtendra el resultado de la
8  petición realizada por el cliente*/
9
10 interface Respsql{
11    void resultado (in string cons);
12
13 };
14
15 /*Esta interfaz enviara sentencias
16 SQL al componente CCM_SQL*/
17
18 interface Envsql{
19
20  void consulta (in Respsql obj, in string cons);
21
22 };
```

A.3.2 Componente cliente

```
1
2  /*El siguiente código es la implementación
3  del cliente CORBA CCM*/
4
5  /* Se implementan las bibliotecas que
6  el compilador IDL genera, para poder
7  realizar la interacción con el componente
8  CCC_SQL*/
9
10 #include "cadsql.h"
11 #include "ccm.h"
12
13 #include <iostream>
14 #include <fstream>
15 #include <unistd.h>
16 #include <unistd.h>
17 #include <stdio.h>
18 #include <string.h>
```

```
19 #include <CORBA.h>
20 #include <coss/CosNaming.h>
21
22 using namespace std;
23
24 /*Se declaran las variables que
25 almacenan los resultados de las
26 sentencias SQL ejecutadas*/
27
28 CORBA::String_var resul;
29 Respsql_ptr respsql = Respsql::_nil();
30 CORBA::Object_var obj;
31
32 /*Implementamos el componente
33 mediante el CcmHome*/
34 CcmHome_var hh;
35 CcmW_var hw;
36
37
38 /*Esta clase envia al cliente el resultado
39 de la sentecia SQL, que éste ha enviado
40 previamente*/
41
42 class Servid : public MICOMT::Thread
43 {
44 public:
45     void _run(void*)
46     {
47             cout << "Enviado␣resultado:␣‘‘" << resul <<
48 "’’al␣cliente." << endl;
49             respsql->resultado(resul);
50 #ifndef _WIN32
51             sleep(1);
52 #else // _WIN32
53             _sleep(1);
54 #endif // _WIN32
55     }
56 };
57
58
59 // Implementacion de la interfaz Envsql
60 /*En esta clase el cliente envia al
61 servidor una consulta SQL*/
62
63 class Envsql_impl:virtual public POA_Envsql
64 {
```

```
65 public:
66     void consulta(Respsql_ptr obj, const char* cons)
67     {
68
69      cout << "El␣cliente␣envio␣la␣consulta"<<
70              cons << "''" << endl;
71
72
73   hw = hh->create ();
74   hw->message(cons);
75
76   /*se envia la consulta al servidor*/
77   hw->enviarConsulta();
78   hw->remove ();
79
80         respsql = Respsql::_duplicate(obj);
81         resul = cons;
82     resul="Consulta␣Realizada";
83         Servid* servid = new Servid();
84         servid->start();
85     }
86 };
87
88 int
89 main (int argc, char *argv[])
90 {
91
92 CORBA::ORB_var orb = CORBA::ORB_init (argc, argv);
93 /*Obtenemos la referencia de la localización del servidor de
94 Componentes CORBA mediante el servicio de nombres */
95
96 obj = orb->resolve_initial_references ("NameService");
97
98   CosNaming::NamingContextExt_var nc =
99   CosNaming::NamingContextExt::_narrow (obj);
100
101     CORBA::ORB_ptr orb2 =
102           CORBA::ORB_init(argc, argv, "mico-local-orb");
103
104 /*Obtenemos la referencia de la localización del servidor CORBA
105 mediante el Portable Object Adapter*/
106
107     CORBA::Object_ptr obj2 =
108           orb2->resolve_initial_references("RootPOA");
109     PortableServer::POA_ptr poa =
110           PortableServer::POA::_narrow(obj2);
```

```
111     assert(!CORBA::is_nil(poa));
112
113 /*Se implementa el objeto Envsql_imp, que permite realizar el
114 envio de la consulta al servidor por parte del cliente */
115
116     Envsql_impl* servant = new Envsql_impl;
117     PortableServer::ObjectId_var id =
118           poa->activate_object(servant);
119
120     PortableServer::POAManager_ptr manager =
121            poa->the_POAManager();
122     manager->activate();
123
124
125  assert (!CORBA::is_nil (nc));
126   /*Encontramos la localización del componente
127    mediante el CcmHome*/
128   obj = nc->resolve_str ("CcmHome");
129   assert (!CORBA::is_nil (obj));
130   hh = CcmHome::_narrow (obj);
131    orb2->run();
132    CORBA::release(manager);
133    CORBA::release(poa);
134    CORBA::release(orb2);
135
136   return 0;
137 }
```

A.3.3 Componente conexion

```
 1 /*Implementamos las bibliotecas generadas por el
 2 compilador IDL para Ccm_SQL*/
 3
 4 #include "libpq-fe.h"
 5 #include "ccm.h"
 6 #include <stdio.h>
 7 #include <string.h>
 8 #include <iostream>
 9 #include <fstream>
10
11 using namespace std;
12
13 /*Varaibles para establecer la
14 conexión a la base de datos*/
15
16 char *hostname = "148.247.102.23";
17 char *port     = "5432";
```

```
18 char *dbname   = "comidas";
19 char *username = "postgres";
20 char *password = "postgres";
21 char *resultado="JAIR";
22 PGconn *pgconn;
23
24 /* establece conexión y
25 autentificación a la BD. */
26
27 PGconn * sql_conn(void){
28         PGconn *conn;
29
30         /* conectar */
31         conn = PQsetdbLogin(hostname,
32                         port,
33                         NULL,  /* options */
34                         NULL,  /* tty */
35                         dbname,
36                         username,
37                         password);
38
39         /* verificar el estado de la conexión,
40            en caso de que ésta falle, envia
41            un mensaje de error, describiendo
42            cual fue el problema y cierra el
43            canal de conexión abierto.
44         */
45         if (PQstatus(conn) != CONNECTION_OK)
46         {
47            fprintf(stderr, "la␣conexión␣a␣%s␣falló.\n", dbname);
48            fprintf(stderr, "%s", PQerrorMessage(conn));
49            PQfinish(conn);
50            exit(1);
51         }
52
53         /* si la conexión funciona se puede
54            continuar.
55         */
56         return conn;
57 }
58
59
60
61 /*
62  *  Recibe una conexión y una sentencia SQL,
63  *  la ejecuta y despliega el resultado en
```

```
64   *  la pantalla.
65   *    */
66  int
67  sql_exec(PGconn *conn, const char *sql)
68  {
69          PGresult   *res;
70
71          int         nfields;
72          int         nrows;
73          int         i, j, l;
74          int        *length;
75
76
77          /* Ejecuta la consulta */
78          res = PQexec(conn, sql);
79
80          /* Verifica el código de retorno */
81          if (!res || PQresultStatus(res) > 2)
82          {
83                  fprintf(stderr,
84  "falló␣la␣consulta:␣%s\n", PQerrorMessage(conn));
85                  fprintf(stderr,
86   "la␣consulta␣era:␣%s\n", sql);
87
88          }
89
90          /* Obtiene el número de campos */
91          nrows = PQntuples(res);
92          nfields = PQnfields(res);
93
94          /* Para cada columna, obtiene el ancho necesario */
95          length = (int *) malloc(sizeof(int) * nfields);
96          for (j = 0; j < nfields; j++)
97                  length[j] = strlen(PQfname(res, j));
98
99          for (i = 0; i < nrows; i++)
100         {
101                 for (j = 0; j < nfields; j++)
102                 {
103                    l = strlen(PQgetvalue(res, i, j));
104                    if (l > length[j])
105                            length[j] =
106                            strlen(PQgetvalue(res, i, j));
107                 }
108         }
109
```

```
110
111         /* imprime cada fila */
112         for (i = 0; i < nrows; i++)
113         {
114           for (j = 0; j < nfields; j++)
115             fprintf(stdout, "%*s", length[j] + 2,
116                             PQgetvalue(res, i, j));
117             fprintf(stdout, "\n");
118
119         }
120
121         /* libera memoria */
122         PQclear(res);
123         free(length);
124
125         return 0;
126 }
127
128
129 /*Esta clase implementa la componente cliente
130 el cual podra responder a cualquier consulta
131 que sea enviada por un cliente */
132
133 class Ccm_impl : virtual public CCM_ {
134
135 private:
136   CORBA::String_var _message;
137
138
139 public:
140   Ccm_impl (const char * initial)
141     {
142       _message = initial;
143     }
144
145 /*consulta que el componente
146 cliente envia al componente
147 servidor.*/
148   void enviarConsulta ()
149     {
150      pgconn = sql_conn();
151      sql_exec(pgconn, _message);
152       cout << _message << endl;
153     }
154
155   void message (const char * val)
```

```
156     {
157       _message = CORBA::string_dup (val);
158     }
159
160   char * message ()
161     {
162       return CORBA::string_dup (_message);
163     }
164 };
165
166 /*Implementación del componente mediand el
167 CCM_Home*/
168
169 class Ccm_impl : virtual public CCM_Home {
170 private:
171   CORBA::String_var _initial_message;
172
173 public:
174   CcmHome_impl ()
175     {
176       _initial_message = CORBA::string_dup ("Sentencia␣SQL");
177     }
178
179   Components::EnterpriseComponent_ptr create ()
180     {
181       return new CcmW_impl (_initial_message);
182     }
183
184   void initial_message (const char * val)
185     {
186       _initial_message = CORBA::string_dup (val);
187     }
188
189   char * initial_message ()
190     {
191       return CORBA::string_dup (_initial_message);
192     }
193 };
194
195 extern "C" {
196   Components::HomeExecutorBase_ptr create_CcmHome ()
197   {
198     return new CcmHome_impl;
199   }
200 }
```

Apéndice B

Vistas del sistema

A continuación se presentan las pantallas (vistas para el usuario) que componen el sistema, el desarrollo en cuanto a funcionamiento e interfaz de usuario es el mismo para los tres casos de middleware estudiados, es decir, que los tres sistemas funcionan con las mismas interfaces, cada una implementada por supuesto con la tecnología que le corresponde, así pues para el caso de DCOM la interfaz se desarrollo con páginas ASP, para EJB se utilizan páginas JSP y en el caso de CORBA queda a gusto del programador, en nuestro caso se utilizaron páginas JSP.

Figura B.1: Pantalla principal del sistema

Figura B.2: Pantalla de ventas

Figura B.3: Pantalla de venta a un cliente

Figura B.4: Pantalla de autentificación del administrador

Figura B.5: Pantalla de altas, bajas y modificaciones

Figura B.6: Pantalla de alta de una pizza

Figura B.7: Pantalla del cálculo del costo de una pizza

Figura B.8: Pantalla de acceso a modificación de una pizza

Figura B.9: Pantalla de modificación de una pizza

Referencias

[1] Aabernethy, R. *COM/DCOM Unleashed*. Macmillan Computer Publishing, 1999.

[2] Armstrong, E., Ball, J., Bodoff, S., Carson, D. B., Evans, I., Green, D., Haase, K., and Jendrock, E. *The J2EE 1.4 Tutorial*. Sun Microsystems, 2006.

[3] Bill Burke, R. M.-H. *Enterprise JavaBeans 3.0*, fifth edition ed. O'Reilly, 2006.

[4] Bolton, F. *PURE CORBA*. Sams Publishing, 2002.

[5] Bond, M., Haywood, D., Law, D., Longshaw, A., and Roxbirgh, P. *Teach Yourself J2EE in 21 Days*. Sams Publishing, 2002.

[6] Brose, G., Vogel, A., and Duddy, K. *Java Programming with CORBA, Advanced Techniques for Building Distributed Applications*, third edition ed. Wiley Publishing, Inc, 2001.

[7] Brown, A. W., and Wallnau, K. C. The current state of cbse. *IEEE Software*, pp. 37–46, September–October 1998.

[8] Cadman, J. *Waite Group's COM/DCOM Primer Plus*. Macmillan Computer Publishing, 1998.

[9] Coulouris, G., Dollimore, J., and Kindberg, T. *Distributed System Concepts and design*. Addison - Wesley Publishing Company, 2003.

[10] D, V., T, P., and A, T. Distributed technologies corba, enterprise javabeans, web services: a comparative presentation. parallel, distributed, and network-based processing. *Parallel Distributed and Network-Based Processing 14th Euromicro International Conference*, No. 5, pp. 103–123, February 2006.

[11] dpunkt verlag. *MICO An Open Source CORBA 2.3 Implementation*. Morgan Kaufmann Pubblishers, Inc., 2005.

[12] Eddon, G., and Eddon, H. *Inside COM+ Base Service*. Microsoft Press, 1999.

[13] Emmerich, W. Software engineering and middleware: A roadmap. *International Conference on Software Engineering,Proceedings of the Conference on The Future of Software Engineering ACM*, pp. 117–129, 2000.

[14] Emmerich, W., and Kaveh, N. Component technologies: Java beans, com, corba, rmi, ejb and the corba component model. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference*, pp. 691 – 692, 2002.

[15] Everett, W. W. Software component reliability analysis. *Application-Specific Systems and Software Engineering and Technology, 1999. ASSET'99 Proceedings. 1999 IEEE Symposium on*, pp. 204–211, 1999.

[16] Gokhale, A., Schmidt, D. C., Natarajan, B., and Wang, N. Applying model-integrated computing to component middleware and enterprise applications. *The Communications of the ACM Special Issue on Enterprise Components, Service and Business Rules 45*, 2002.

[17] Group, O. M. *The common Object Request Broker: Architecture and Specificaction revision 3.0*. OMG, 2004.

[18] Group, O. M. *CORBA Component Model Specification OMG Available Specification Version 4.0*. OMG, 2006.

[19] III, F. E. R. *DCOM:Microsoft Component Object Model*. IDG Books Worldwide, Inc, 1997.

[20] Irene Luque Ruiz y Miguel Ángel Gómez-Nieto, Enrique López Espinosa, G. C. G. *Bases de Datos Desde Chen hasta Codd con ORACLE*. Alfaomega Ra-Ma, Pitágoras 1139, Col. Del Valle, 03100 México, D.F, 2002.

[21] J., C. Middleware move to the forefront. *IEEE Computer 32*, No. 5, pp. 17–19, May 1999.

[22] Juric, M. B., and Rozman, I. Choosing component middleware based on performance evaluation, 2000.

[23] Kotonya, G., Sommerville, I., and Hall, S. Towards a classification model for component-based software engineering research. *29th EUROMICRO Conference "New Waves in System Architecture", IEEE Computer Society* pp. 1–10, 2003.

[24] Loualalen, M., Moreaux, P., and Salmi, N. Structured analysis for component-based systems: an ejb/corba application. *First International Workshop on Verification and Evaluation of Computer and Communication Systems*, May 2007.

[25] Michi Henning, S. V. *Advanced CORBA Programming with C++*. Addison Wesley, 1999.

[26] Mojica, J. *COM+ Programming with Visual Basic*. O'Reilly, 2001.

[27] Pattinson, T. *Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0*. Microsoft Press, 2000.

[28] Paul J. Perrone, V. S. R., Chaganti, K. R., and Schwenk, T. *J2EE Developer's Handbook*. Sams Publishing, 2003.

[29] Peishu, L. *Visual Basic and COM+ Programming by Example*. Prentice Hall - QUE, 2002.

[30] Plasil, F., and Stal, M. An architectural view of distributed objects and components in corba, java rmi, and com/dcom. *A submission to Sotfware-Concepts and Tools, Springer 66*, No. 21, 1998.

[31] Pressman, R. S. *Software Engineering A. Practitioner's Approach*. McGraw-Hill, 2005.

[32] Puder, A., Römer, K., and Pilhofer, F. *Distributed Systems Architecture A Middleware Approach*. Morgan Kaufmann Publishers and Elsevier, 2006.

[33] Rima Patel Sriganesh, Gerald Brose, M. S. *Mastering Enterprise JavaBeans 3.0*. Wiley Publishing, Inc, 2006.

[34] Rubin, W., and Brain, M. *Understanding DCOM*. Prentice Hall, Inc., 1999.

[35] Sanghera, P. *SCBCD Exam Study Kit, Java Business Component Developer Certification For EJB*. Manning Publicactions Co., 209 Bruce Park Avenue, Greenwich, CT 06830, USA, 2005.

[36] Silberschat, A., and Baer, P. *Operating Systems Concepts*. Addison - Wesley Publishing Company, 1999.

[37] S.N., F., T.B., Q., Oapos, Connor, M., Mulcahy, B., and Morrison, J. A framework for heterogeneous middleware security. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pp. 108 – 118, April 2004.

[38] Sommerville, I. *Software Engineering*. Pearson Education, 2005.

[39] Szyperski, C., Gruntz, D., and Murer, S. *Component Software Beyond Object-Oriented Programiming*. Addison - Wesley Publishing Company, 2001.

[40] Tapadiya, P. *COM+ Programming: A Practical Guide Using Visual C++ and ATL*. Prentice Hall, 2000.

[41] Wang, A. J. A., and Qian, K. *Component-Oriented Programming*. Wiley & Sons, 2005.

[42] Zarras, A. A comparison framework for middleware infrastructures. *In Journal of Object Tecnology 3*, No. 5, pp. 103–123, June 2004.

Referencias Web

[43] http://java.sun.com/javaee.

[44] http://sourceforge.net/projects/cif/.

[45] http://www.cpi.com/ejccm.

[46] http://www.cse.wustl.edu/ schmidt/CIAO.html.

[47] http://www.omg.org.