Universidad Autónoma de Querétaro.

Facultad de Informática.

Maestría en Ingeniería de Software Distribuido.

Especificación de una Interfaz Común SQL para la comunicación entre bases de datos heterogéneas en diferentes sistemas operativos

TESIS

Que como parte de los requisitos para obtener el grado de:
"Maestro en Ingeniería de Software Distribuido"

Presenta:
ISC. Christian Mirsha Velasco Castilla

Dirigido por:
MSI. Gerardo Rodríguez Rojano

SINODALES

MSI. Gerardo Rodríguez Rojano
Presidente

Firma

MISD. Juan Salvador Hernández Valerio
Secretario

Firma

MC. Alberto Lamadrid Álvarez
Vocal

Firma

MC. Ruth Angélica Rico Hernández
Suplente

Firma

MISD. Carlos Alberto Olmos Trejo
Suplente

Firma

MC. Alejandro Santoyo Rodríguez
Director de la Facultad de Informática

Dr. Luis Gerardo Hernández Sandoval
Director de Investigación y Posgrado

Centro Universitario
Santiago de Querétaro, Querétaro
Marzo 2008
México

Resumen

Ubicado en un contexto de Sistemas Distribuidos, este trabajo de investigación presenta y describe las especificaciones técnicas que conforman a un componente denominado “Interfaz Común SQL”. Se ubica en la capa intermedia de la arquitectura de aplicaciones Cliente/Servidor proporcionando una forma única de comunicación e interacción entre aplicaciones destinadas para el usuario final con diversos Sistemas para la Administración de Base de Datos, los que proporcionan el soporte tecnológico y heterogéneo de una base de datos distribuida. Para lograrlo, se realizó una revisión a los diversos temas que engloban a esta área de la Informática y que dan un panorama amplio del contenido de la Maestría en Ingeniería de Software Distribuido que se imparte en la Universidad Autónoma de Querétaro, incluidos el Lenguaje Unificado de Modelado, la Programación Orientada a Objetos y el Lenguaje de Programación Java, con los que se realizó y documentó este prototipo. Con estas herramientas, se muestra su estructura estática y el comportamiento dinámico que adquiere dentro de un marco de trabajo en donde se requiere obtener información y manipular datos de manera remota, aprovechando los recursos tecnológicos de los que disponen las organizaciones actuales. Se ofrece soporte a Microsoft SQL Server 2000 Desktop Edition, a MySQL Database Server versión 4.0.20 y a Oracle 9i Database Server versión 9.0.1, que se usan como las plataformas de gestión de datos elegidas para probar el desenvolvimiento de este elemento. Además, se proporcionan una serie de recomendaciones para el control de la heterogeneidad que, utilizándolas en la creación de nuevas aplicaciones basadas en esta tecnología, consigue sistemas compatibles, integrados y uniformes. Por último, se incluye el código fuente y los esquemas de configuración externa de este middleware para mayor referencia de su funcionamiento.

(Palabras clave: Middleware, DBMS, Federación, Heterogéneo)

Summary

Located in a context of Distributed Systems this investigation presents and describes the technical specifications of a component named Common Interface of SQL. It is located in the interlayer of the applications architecture Server/Client providing a unique form of communication and interaction among destined applications for the end user with diverse Database Management Systems, which provides the technological and heterogeneous support give a distributed database. To achieve that, it was carried out a revision to the sundries topics that they include to this area of Computer science and that they give a wide panorama of contents gives the Master in Engineering of Distributed Software that is imparted in the Universidad Autónoma de Querétaro, including the Unified Modeling Language, the Object Oriented Programming and the Java Programming language, with those built and documented prototype. With these tools, is shown their static structure and the dynamic behavior that acquires inside a framework where if required to obtain information and to manipulate data in a remote way, taking advantage of the technological resources gives that preparation to the current organizations. This document offers support to Microsoft SQL Server 2000 Desktop Edition, to MySQL Database Server version 4.0.20 and to Oracle 9i Database Server version 9.0.1 that is used as the platform give data management chosen to prove the performance of this element. Also, they are provides a series of recommendations for controlling heterogeneity used in the creation of new applications based on this technology, it gets compatible, integrated and standardize systems. Lastly, are included the source code and the schemes of external configuration of this middleware for further reference of their operation.

(Keywords: Middleware, DBMS, Federation, Heterogeneous)

Dedicatorias

Al Padre Universal que es la base de todo, el origen de todas las causas y efectos, aunque no entienda sus decisiones, sé que tiene una razón válida y sabia para que aprenda a lograr el equilibrio interno de mi Ser en todo momento.

Con toda mi admiración, agradecimiento y cariño a mi Papá Luis Modesto Velasco Mota por apoyarme en todo momento que lo he requerido, por las palabras de aliento y ánimo, por decirme no lo que quiero escuchar, sino lo que necesito saber, y por incluirme en la realización de sus sueños.

A mi Mamá Rosario Castilla Flores porque gracias a su Amor, a sus consejos, a sus atenciones, su interés en mí y en todo lo que me ha rodeado, a su comprensión y apoyo incondicional puesto que hemos logrado juntos la culminación de una etapa más de nuestra vida.

A mi Hermana Liliana Jacqueline Velasco Castilla, porque gracias a su entusiasmo, a su alegría, su apoyo fraternal, sus opiniones e inquietudes y a su hombro en donde sostenerme cuando más lo he necesitado, me ha dado nuevos bríos, fortaleza y paciencia para completar el logro de nuestros objetivos.

A mi Familia que con sus alegrías, muestras de cariño y afecto me han dado una razón más para festejar la vida con ellos, para disfrutarla y para compartirla.

A mi hermosa preciosa, Cinthia Nangeli Delgado Arriaga por ser una luz en mi camino, por dedicarme su Amor, cariño, ternura, tiempo, palabras y paciencia que me han dado una gran alegría y entusiasmo para conducirnos a una nueva realidad.

Agradecimientos

Con mucho gusto y satisfacción le doy las gracias al MSI. Gerardo Rodríguez Rojano por cada una de las ocasiones en las que me presenté ante él para atender sus recomendaciones y orientaciones para el desarrollo de este trabajo de investigación. Le agradezco su paciencia y su comprensión hacia este alumno que únicamente ha aprendido de usted, con el paso del tiempo, ha ser más responsable, más dedicado y más “picudo” en sus razonamientos y conclusiones.

Nuevamente muchas gracias por todo… Maestro.

De igual manera, quisiera extender estos agradecimientos al MC. Alberto Lamadrid Álvarez por su sincera expresión de alegría y satisfacción, a este trabajo de investigación.

También quiero agradecer las recomendaciones que me proporcionaron para mejorar e incrementar la calidad de este trabajo de investigación a el MISD. Juan Salvador Hernández Valerio y al MISD. Carlos Alberto Olmos Trejo, quienes atenta y gustosamente atendieron a todas y cada uno de mis cuestionamientos e inquietudes.

Quisiera ofrecerle mi reconocimiento, admiración y gratitud a la MC. Ruth Angélica Rico Hernández por sus valiosos comentarios e indicaciones durante mi proceso de formación.

Al Consejo Nacional de Ciencia y Tecnología, CONACYT por haberme proporcionado la confianza y los recursos necesarios para darle a mi país, a mis seres queridos y a mí, la satisfacción de haber alcanzado un nuevo escalón en el continuo aprendizaje del ser humano y lograr completar la Maestría de Ingeniería de Software Distribuido de la Universidad Autónoma de Querétaro.

Índice

Página

Página

Índice de cuadros

Índice de figuras

Figura		Página
2.26	Diagrama de actividades.	77
2.27	Modelo de dos capas JDBC.	80
2.28	Modelo de tres capas JDBC.	81
2.29	Puente JDBC-ODBC.	82
2.30	Controladotes Java/Binario.	83
2.31	Controlador 100% Java/Protocolo nativo.	84
2.32	Controlador 100% Java/Protocolo independiente.	85
3.1	Arquitectura Cliente/Servidor e Interfaz Común SQL.	90
3.2	Ambiente heterogéneo y distribuido de base de datos.	91
3.3	Arquitectura de referencia de una base de datos distribuida.	92
3.4	Esquema global de una base de datos distribuida.	93
3.5	Esquema de reparto.	95
3.6	Interfaz Común SQL y operaciones con base de datos.	97
3.7	Estructura de la Interfaz Común SQL y módulos principales.	100
3.8	Despliegue de la Interfaz Común SQL.	100
3.9	Actividades generales de la Interfaz Común SQL.	101
3.10	Vista estática de la Interfaz Común SQL.	102
3.11	Estado general de la Interfaz Común SQL durante su ejecución.	104
3.12	Diagrama de secuencia para Actualizar.	106
3.13	Diagrama de secuencia para Agregar.	107
3.14	Diagrama de secuencia para Borrar.	108
3.15	Diagrama de secuencia para Buscar.	109
3.16	Funcionalidad de la clase Archivo.	110
3.17	Actividades de la clase Archivo.	110
3.18	Actividades de la clase Tablas.	111
3.19	Funcionalidad de la clase Generador.	113
3.20	Actividades de la clase Generador.	114
3.21.	Actividades para generar la instrucción DELETE.	118
3.22	Actividades para generar la instrucción INSERT.	119

Figura		Página

I. INTRODUCCIÓN

I.1. Descripción del problema.

Información es una palabra común en todas las organizaciones actuales. El Diccionario de la Lengua Española (Real Academia Española, 2001) define la palabra Dato como: "Hechos conocidos que pueden registrarse y que tienen un significado implícito" y también como: "Antecedente necesario para llegar al conocimiento exacto de algo o para deducir las consecuencias legítimas de un hecho". Los datos al ser analizados y cuestionados se les proveen de un significado que produce información y conocimiento, lo cual es necesario para la toma de óptimas decisiones a cualquier nivel sobre cualquier situación que se presente. Actualmente los datos, o la descripción parcial o total sobre cualquier objeto de análisis que tenga importancia en la vida cotidiana y laboral, son registrados, administrados y compartidos con otras personas u organizaciones por sistemas de cómputo.

Los sistemas de cómputo son una combinación de *hardware*[1] y *software*[2] que trabaja de manera conjunta, sincronizada e integrada para la correcta disposición y manejo de los recursos que proporciona para la administración y resguardo de los datos. El ambiente de competencia, la necesidad de conseguir nuevos clientes y mantener los existentes a través del ofrecimiento de mejores productos y servicios entre las empresas que los utilizan, hace que exista una gran diversidad en la construcción de los sistemas computacionales, aumentando la complejidad, que los estándares informáticos establecidos sean modificados a la conveniencia de cada una de ellas y que el intercambio de datos e información entre los ambientes operativos sea cada vez más difícil. Los elementos de *hardware*, los sistemas operativos, las comunicaciones en red y las aplicaciones de usuario así como las dedicadas a la gestión de los datos, son componentes de un sistema computacional y están propensos a esta situación.

[1] Conjunto de componentes que integran la parte material de una computadora. (Real Academia Española, 2001).
[2] Conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora. (Real Academia Española, 2001).

El establecimiento de estándares computacionales como el Lenguaje Estructurado de Consultas que utilizan los Sistemas de Gestión de Bases de Datos en general, ha ayudado al intercambio de información de manera transparente para el usuario final y para las aplicaciones que lo utilizan. Sin embargo, toda herramienta de *software* es propensa a modificaciones, lo que hace que existan diferentes versiones de éste y que cada uno de los sistemas mencionados realicen, además, ampliaciones al mismo, dando como resultado un conjunto de combinaciones entre estos elementos lo que aumenta la incompatibilidad en los procesos de comunicación e intercambio de datos.

El "tipo de dato", es quizá el concepto más importante que existe en el mundo de las bases de datos puesto que ellos, son los responsables de asemejarse lo más fielmente posible a la realidad que se requiere almacenar dentro de un Sistema de Base de Datos, sin embargo, cada fabricante de estas herramientas también tienen su propia perspectiva sobre como representarlos e interpretarlos y, aunque es válido, esta situación conlleva a incompatibilidades y pérdida de significado en la transferencia de los mismos.

Las grandes empresas o cualquier otra agrupación actualmente se ayudan de los Sistemas de Bases de Datos para la toma de decisiones importantes que les afectan en su productividad. Al crecer éstas en su complejidad también lo hace la cantidad de datos a gestionar por estos sistemas y obtener información por lo que, la distribución de todo ese conjunto de activos se hace indispensable; normalmente este proceso refleja la organización de las mismas. Un Sistema de Base de Datos Distribuida es el concepto útil para tal fin, sin embargo, cuáles *software* de gestión y cómo integrarlos para dar la apariencia de un sistema único o centralizado, hace que esta elección deba de hacerse con cuidado, en el caso de utilizar un ambiente heterogéneo de cómputo.

El presente documento de investigación: "Especificación de una Interfaz Común SQL para la comunicación entre bases de datos heterogéneas en diferentes sistemas operativos" propone una serie de principios de diseño que requiere un *middleware* que haga posible la obtención, la manipulación y el intercambio de datos que contiene una base de datos distribuida en distintos lugares y administrada por diferentes aplicaciones gestoras de datos. Para lograrlo, es necesario revisar los componentes principales y herramientas necesarias de los Sistemas Distribuidos de Bases de Datos Relacional que permiten la construcción y utilización de una base de datos en general, los conceptos necesarios para

colocarla en un entorno de computación distribuida y las herramientas existentes para modelar las especificaciones de nuevos componentes de *software* dentro de un ambiente operativo que muestre las condiciones que actualmente imperan en las organizaciones.

I.1.1. Hipótesis.

Una “Interfaz Común SQL” permite que las aplicaciones Cliente se comuniquen, actualicen datos y obtengan información de distintos Servidores que soportan a una base de datos distribuida y heterogénea.

I.2. Objetivos de la investigación.

I.2.1. Objetivo general.

Demostrar que mediante una investigación y elección adecuada de los conceptos, técnicas, lineamientos y herramientas que corresponden a la arquitectura de aplicaciones Cliente/Servidor se puede lograr la “ilusión de un sistema único” en los sistemas distribuidos y heterogéneos de bases de datos.

I.2.2. Objetivos específicos.

- Revisar los temas y conceptos básicos que dan forma a los Sistemas Distribuidos y heterogéneos de base de datos para obtener un sustento científico y válido en el desarrollo del presente trabajo de investigación.
- Elegir las técnicas y herramientas más adecuadas para el diseño de un componente *middleware* y exponer su ámbito de operación dentro de la arquitectura de aplicaciones Cliente/Servidor.
- Demostrar que mediante un uso disciplinado y sustentado del Lenguaje Unificado de Modelado se puede documentar, de manera adecuada, el proceso

de Ingeniería de *Software* realizado en la construcción de Sistemas Distribuidos.

- Comprender cómo afecta la heterogeneidad de los componentes que forman a las aplicaciones Cliente/Servidor y cómo se aplican las técnicas elegidas para el logro de la interoperabilidad en los mismos.

I.3. Justificación.

Sobre la problemática que existe en la integración, comunicación e interacción de las aplicaciones Cliente con distintos Servidores de una base de datos distribuida y heterogénea, Robert Orfali, Dan Harkey y Jeri Edwards en su libro “Cliente/Servidor. Guía de supervivencia”, determinan que con un *middleware* específico para base de datos y el utilizar un conjunto de servidores de base de datos de proveedores múltiples, propiedad autónoma y acoplamiento holgado que se comuniquen utilizando una modalidad llamada Sistemas de Base de Datos en Federación, se puede tener una solución.

El *middleware* debe de proporcionar un conjunto de interfaces por medio de las cuales, una aplicación Cliente invoca un servicio que transmita la solicitud por la red de computadoras hasta llegar a los Servidores que soportan a la base de datos distribuida para obtener la información que se requiere o manipular los datos que la componen. Sin embargo, esta alternativa es únicamente válida para entornos heterogéneos y distribuidos de base de datos.

El uso de sistemas conformados por un solo proveedor para la gestión de los datos repartidos en varios sitios de una red, situación denominada por los autores como “Nirvana de SQL”, sería una opción más adecuada, sin embargo, en la actualidad disponemos de una gran cantidad de herramientas para integrar un Sistema Distribuido de cómputo. Las organizaciones incluso ya tendrán cierta tecnología adquirida por lo que recomendarles que las desechen y adquieran nuevas soluciones enfocadas a un paradigma, repercutiría en el incremento de sus costos, algo muy complicado de enfrentar por los directivos que necesitan aprovechar al máximo cada inversión realizada con anterioridad.

En (Orfali, et. al. 1998) se propone una solución encaminada a lograr el “Nirvana en federación” donde el primer paso es la creación de una “Interfaz Común SQL”. La idea

es crear un componente *middleware* que pueda ser empleada por las aplicaciones Cliente y dejar que las diferencias entre los Servidores de una base de datos heterogénea y distribuida, sean tratadas por los diferentes controladores de acceso a ellas. Figura 1.1.

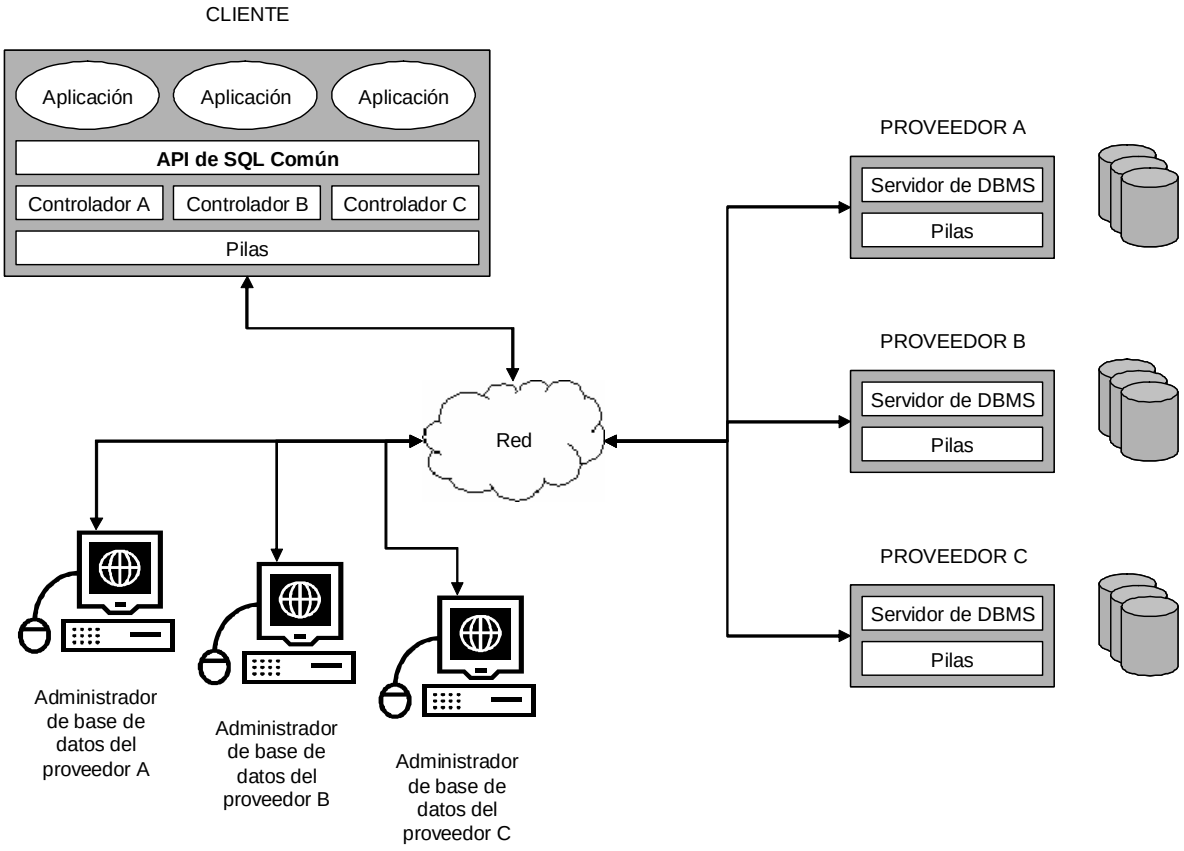

Figura 1.1. Una API de SQL Común.

Para apoyar a las especificaciones de una “Interfaz Común SQL” se aprovecha el uso de los diagramas del Lenguaje Unificado de Modelado que proporcionan las vistas estática y dinámica del mismo. Una de las aportaciones de este documento de investigación, es la exposición de que la utilización disciplinada de cada uno de ellos permite expresar el diseño de cualquier sistema para que la comunidad pueda leer y entender de una manera gráfica los componentes que lo integran de una manera conceptual.

I.4. El método de investigación.

El enfoque de investigación elegido para el cumplimiento del objetivo general de este documento y la comprobación de la hipótesis que lo origina se le denomina Cuantitativo utilizando el método deductivo. Se parte de conceptos generales aceptados por la comunidad informática como válidos que son sometidos a un proceso de análisis sobre su funcionamiento y posibilidades de interacción entre ellos para que den forma al entorno operativo y a la formulación de las especificaciones de una “Interfaz Común SQL”.

I.5. Estructura del documento.

El capítulo II “Revisión de Literatura” contiene extractos de la literatura actual, referentes al tema, y que se encuentra dividido en varios apartados. Se inicia con la sección “Sistemas Distribuidos” donde se describen los conceptos y las ideas relacionadas con los “Sistemas operativos y protocolos de comunicación en red”, que son la base de funcionamiento de las “Aplicaciones Cliente/Servidor” y que juntos que dan forma a todo el concepto que define a este capítulo. La sección “Sistemas de Base de Datos Relacional” revisa lo que son los sistemas de base de datos, sus características, los modelos necesarios para la especificación de las mismas y su administración. En “Base de distribuidas” se encuentra una revisión de esta tecnología en los ambientes distribuidos de cómputo y el lugar que ocupan dentro de la tecnología Cliente/Servidor en “Servidores de bases de datos relacional”. Por último, la sección “Ingeniería de Software y Programación Orientada a Objetos” describe los procedimientos para el análisis y diseño de una aplicación orientada a objetos por medio del uso de los diagramas del “Lenguaje Unificado de Modelado” y la utilización del lenguaje de programación Java y su interacción con los sistemas de administración de bases de datos en “El lenguaje de programación Java y bases relacionales de datos”.

El capítulo III “Metodología” y en sus diferentes secciones se encuentra la descripción de cómo se aplicaron los diferentes e importantes fundamentos teóricos para la obtención de los principios de diseño de la “Interfaz Común SQL”, de sus componentes principales y su funcionamiento dentro de un “Campo de acción experimental”.

El capítulo IV “Resultados y Discusión” plantea las conclusiones relacionadas a esta investigación y presenta las observaciones surgidas durante el proceso de diseño de la “Interfaz Común SQL”. Además propone nuevos trabajos futuros dentro de este ramo de la computación distribuida.

El “Apéndice” en sus diferentes secciones contiene el código fuente del prototipo creado para esta investigación y los formatos de los archivos de configuración externos que requiere.

II. REVISIÓN DE LITERATURA

II.1. Sistemas Distribuidos.

Un sistema es un conjunto de elementos interrelacionados entre sí que a través de la aplicación de ciertos procesos predefinidos a un conjunto de datos del entorno o ambiente en el que se desenvuelve, persigue el lograr objetivos establecidos con anterioridad (Kendall y Kendall, 1997). Estos procesos son llevados a cabo por los llamados "subsistemas", elementos o sistemas más pequeños que no interfieren para funcionar entre sí pero que el resultado de cada uno de ellos sirve para generar u obtener el resultado que el sistema en general necesita; vea Figura 2.1. Los "Sistemas de Cómputo" son un conjunto de subsistemas, llamados *hardware* y *software* que trabajan juntos para lograr los objetivos para los cuales fueron diseñados.

Las actuales soluciones informáticas de las organizaciones se encuentran muy relacionadas con este enfoque, sin embargo, desde los inicios de la computación, el procesamiento de datos era de manera centralizada por lo que todos los procesos recaían en la computadora central mientras que los usuarios empleaban sencillas terminales informáticas para comunicarse con la central y obtener información. Los sistemas o las soluciones informáticas centralizadas eran aquellas que se ejecutaban en un único Sistema Operativo[3] u OS (*Operating System*).

Esta forma de organización hizo que el mantenimiento del *hardware* y especialmente de la computadora central o *mainframe* fuese un proceso lento e incluso peligroso puesto que ahí se almacenan todos los datos e información del negocio imprescindibles y un error físico o lógico dentro de sus componentes podría causar la desaparición de los mismos llevando a pérdidas económicas para la empresa. De la misma manera, el mantenimiento del *software* también era complicado puesto que una actualización a las aplicaciones empresariales destinadas al manejo de los datos centralizados implicaba un cambio en todas y cada una de las terminales habilitadas para la

[3] Programa o conjunto de programas que efectúan la gestión de los procesos básicos de un sistema informático, y permite la normal ejecución del resto de las operaciones. (Real Academia Española, 2001).

conexión con la computadora central, consumiéndose bastante tiempo en este procedimiento.

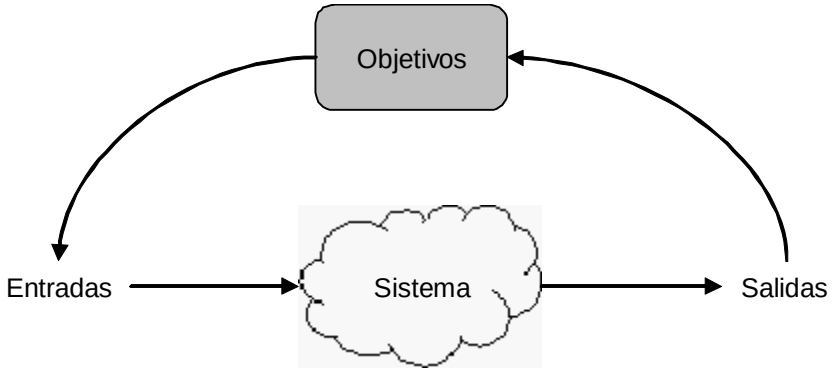

Figura 2.1. Un sistema y sus relaciones.

En un sistema centralizado de cómputo, cuando la computadora central se encontraba en mantenimiento, las terminales no podían acceder a los datos que ésta almacenada. La solución a esta principal dificultad vino de mano de los "Sistemas Distribuidos" que entre otras cosas permite obtener los datos que se encuentran almacenados y distribuidos en varias computadoras conectadas a través de una red de comunicaciones.

Dependiendo del contexto en el que se mencionen, existen diferentes nombres para referirse a las computadoras que forman parte de un sistema distribuido, tales como *emplazamientos* o *nodos.* (Silberschatz, et. al., 1998). Para enfatizar la distribución física de estos sistemas se utiliza principalmente el término *emplazamiento* y, normalmente para referirse a la distribución lógica se les llama *nodo*. En la Figura 2.2, se muestra la estructura general de un sistema distribuido.

Los emplazamientos de los sistemas distribuidos pueden encontrarse en una misma sala de cómputo, pueden estar repartidos en varias oficinas dentro de una empresa o incluso en lugares geográficamente distintos y lejanos. Para que dos emplazamientos se comuniquen con éxito, se requiere que hablen el mismo idioma[4]. Lo que se comunica, cómo se comunica y cuándo se comunica debe seguir una serie de convenciones

[4] Se refiere al modo particular de las computadoras para comunicarse entre sí utilizando un mismo código.

mutuamente aceptadas por las entidades involucradas y se denominan “Protocolos de Comunicación”. Un protocolo es el conjunto de reglas o convenios que definen la forma en que dos entidades cooperantes intercambian los datos.

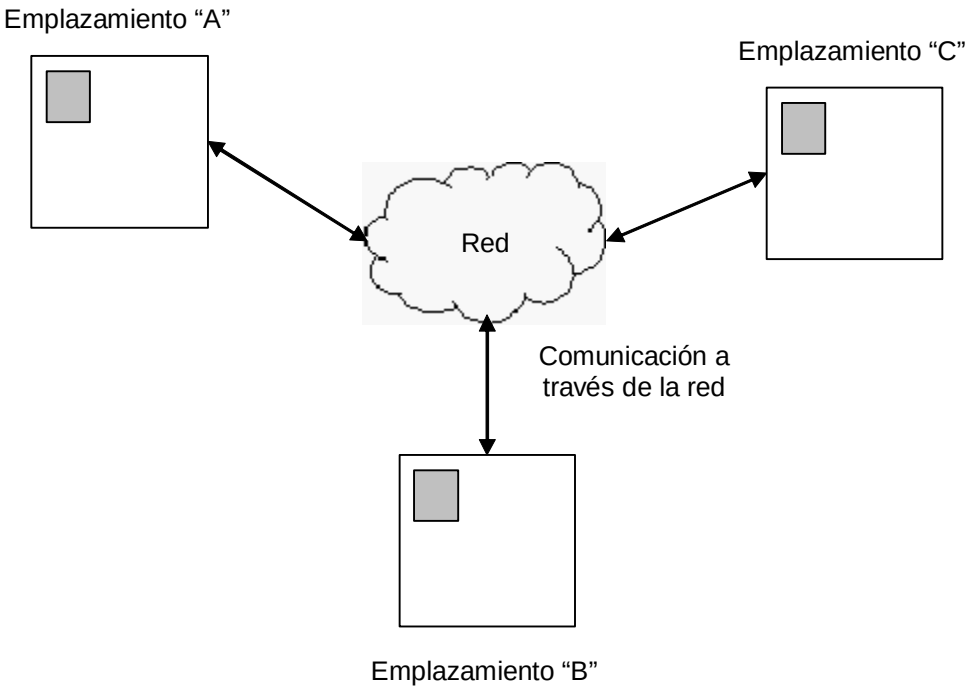

Figura 2.2. Un sistema distribuido.

El conjunto de protocolos TCP/IP, llamado así por la unión de los nombres del “Protocolo para el Control de Transmisión” o TCP (*Transmission Control Protocol*) y del “Protocolo de Red Interna” o IP (*Internet Protocol*), es el más adoptado para la interconexión de sistemas de cómputo. Permite que cualquier par de emplazamientos se comuniquen a pesar de las diferencias físicas del *hardware*, es decir, mantiene la comunicación entre el emplazamiento emisor y el de destino aun si alguna de las computadoras o si los medios de transmisión en el trayecto fueran distintos (Stallings, 1997). Los objetivos de utilizar este conjunto de protocolos dentro de los Sistemas Distribuidos son, entre otros:

- Darle a la red la capacidad de mejorar ante cambios en cualquiera de los elementos de la red, es decir, debe de ser flexible en la distribución de los emplazamientos y evitar detener el acceso a los servicios a todos los terminales conectados por un cambio en la misma.
- Permitir que los sistemas distribuidos con plataformas heterogéneas se conecten a la red y se intercomuniquen entre sí.

Un Sistema Distribuido junto con los Protocolos de Comunicaciones debe dotar transparencia a la constitución de la red para que el usuario final y el programador de aplicaciones no tengan que aprenderse la ubicación de los demás equipos que están conectados a la misma para utilizar un recurso. Debe dar la apariencia de un sistema centralizado. Para realizar lo anterior, el Sistema Distribuido debe estar construido de manera tal que cuando un usuario desee un servicio de la red, la aplicación debe de encargarse de todos los detalles de ejecución de la orden. La tarea de encontrar el servicio, transportar los datos necesarios para obtenerlo y poner los resultados en un lugar adecuado, corresponde al OS y a las aplicaciones destinadas para tal fin.

II.1.1. Sistemas operativos y Protocolos de comunicación en red.

Un Sistema Operativo es un conjunto de programas que efectúan la gestión de los procesos básicos de una computadora y permite la normal ejecución del resto de las operaciones. Es una capa compleja entre el *hardware* y el usuario que le facilita al usuario las herramientas e interfaces adecuadas para realizar sus tareas informáticas, abstrayéndole de los complicados procesos necesarios para llevarlas a cabo. Por ejemplo, un usuario normal simplemente abre los archivos grabados en un disco, sin preocuparse por la disposición de los bits en el medio físico de almacenamiento como los disquetes, CD-ROM, discos duros, tarjetas de memoria portátil, etcétera. Este elemento es un *software* responsable del control o administración del *hardware*. También recibe el nombre de núcleo (conocido en términos informáticos como *kernel*). (Tanenbaum, 1997).

Desde el punto de vista de los Sistemas Operativos, una red de computadoras construida basándose en uno solo se le denomina homogénea y en caso contrario,

heterogénea. En el primer tipo, la complejidad de conectar las computadoras entre sí no es muy difícil ya que se tendrán que seguir una serie de procedimientos que el mismo OS integra para realizarlo. Sin embargo, actualmente existen varias organizaciones en la que es muy fácil encontrar situaciones en donde se requiera utilizar una red de comunicaciones entre computadoras que requieran ejecutar diferentes plataformas, lo cual conlleva a más complejidad en su conexión.

En el caso de aplicaciones Cliente/Servidor (vea la sección II.1.2), correr el mismo Sistema Operativo (red de computadoras homogénea) tanto en clientes como en múltiples servidores simplificaría enormemente la administración de las redes y haría posible la fácil propagación e integración de programas y servicios entre ellos. Además, los procedimientos de instalación, sistemas de archivos e interfaces con el OS serán iguales en este caso para el Cliente y Servidor. Esta situación permite la introducción de aplicaciones con esta arquitectura de diseño en departamentos y pequeñas organizaciones. Sin embargo, para la mayoría de los usuarios no existe el Sistema Operativo único que satisfaga todas estas necesidades de cómputo y, por consiguiente, es posible que dentro de una organización existan diferentes computadoras con diferentes sistemas operativos que desempeñen ciertas funciones importantes para la empresa (Orfali, et. al., 1998).

Los tipos de servicios que un OS puede proporcionar a una red de computadoras y a las aplicaciones Cliente/Servidor que funcionan sobre ella son muy diversos como intercambiar información, transferir archivos, ejecutar comandos remotos, integrar recursos (impresoras, unidades de respaldo, memoria, procesos, unidades centrales de procesamiento), recuperarse de fallas de algunos recursos distribuidos y consolidar la protección y seguridad entre los diferentes componentes del sistema y los usuarios. Para mantener la ilusión de un Sistema Distribuido cada usuario autorizado debe de tener un acceso transparente a cada uno de los elementos del sistema al cual tenga derecho desde el OS.

Para los Sistemas Distribuidos, las funciones del Sistema Operativo se dividen en servicios básicos o complementarios. Los servicios básicos forman parte del sistema estándar, mientras que los servicios complementarios son componentes del *software* modular adicional que funcionan sobre los servicios básicos. Dentro de los servicios básicos de un OS deben de estar la preferencia de multitareas, la prioridad de tareas,

mecanismos de sincronización, comunicaciones entre procesos locales o remotos, unidades de concurrencia, protección entre tareas, un sistema de archivos de alto desempeño para usuarios múltiples, administración eficiente de la memoria y extensiones de tiempo de ejecución de vinculación dinámica. Dentro de los servicios complementarios se pueden encontrar a un conjunto de pilas de protocolos de comunicaciones, servicios para compartir archivos, impresoras y unidades de disco, soporte para Grandes Objetos Binarios o BLOB (*Binary Large OBject*) como imágenes y vídeo, sistemas para localizar servidores y servicios de red disponibles, servicios de autenticación y autorización, administración de sistemas distribuidos, horario de la red o sincronización de relojes, servicios de bases de datos y de transacciones, servicios de Internet y orientados a objetos, entre otros.

Los OS con capacidades de red aportan una de las características más importantes dentro de un Sistema Distribuido, hacer posible la virtual existencia de un sistema único funcionando a partir de otros varios sistemas; sin esa característica, la computación distribuida no sería posible. Transparencia significa hacerle creer a los usuarios que en el sistema distribuido no existen uniones. Significa ocultarles a los usuarios e incluso a los programadores de aplicaciones, tanto la red como sus servidores en diferentes aspectos:

- Transparencia de ubicación: El usuario no necesita conocer la ubicación de un recurso compartido.
- Transparencia de espacio para nombres: El usuario debe estar en la posibilidad de emplear las mismas convenciones de nombramiento de archivos (y espacio para nombres) para localizar cualquier recurso en la red.
- Transparencia de acceso: El usuario debe disponer de la facilidad de dar una sola contraseña (o autenticación) que funcione para todos los servidores y todos los servicios de la red.
- Transparencia de duplicación: Al usuario no le debe de interesar cuántas copias existan de un recurso. A los OS les corresponde sincronizar las actualizaciones y ocuparse de todos los asuntos de seguridad.

Para lograr lo anterior se utiliza un elemento denominado “alias” para ocultar a los usuarios las rutas de localización de recursos dentro de una red de computadoras. De

este modo, los OS extienden transparentemente el soporte de dispositivos del OS local a lo largo de la red. Prácticamente todo lo que puede hacerse en un OS local también se puede hacer en forma remota y transparente. También permiten que aplicaciones generadas en un OS local, se conviertan en aplicaciones en red sin modificar ninguna línea de código. La mayoría de los OS permite que los clientes que operan con otras plataformas diferentes compartan archivos y otros dispositivos.

Este mecanismo de alias a los recursos de la red distribuida y que están propagados se registra dentro del Servicio de Directorio de los OS. Este componente rastrea todos los recursos de los OS y sabe en dónde se encuentra cada uno. Un directorio distribuido debería ofrecer idealmente una imagen única que pudieran usar todas las aplicaciones de red. Normalmente el servicio de directorio se implementa como una base de datos de objetos distribuida y duplicada. Es distribuida, porque permite que diferentes dominios de administración controlen su entorno. Es duplicada, para brindar alta disponibilidad y desempeño donde se necesite. Si el directorio esta fuera de servicio, toda la actividad de la red se detendrá. Es una base de datos de objetos en el sentido de que todo lo que se registra es una instancia de una nueva clase de objetos. Se puede utilizar la herencia para derivar nuevos tipos de objetos. Para el duplicado, un directorio mantiene utiliza una duplicación inmediata que provoca que las copias sean inmediatamente actualizadas con la información del directorio maestro al momento; y utiliza una duplicación periódica, dependiendo del momento elegido, para actualizar todos los cambios realizados en el directorio maestro en las copias.

Mantener una noción del tiempo en la red de computadoras y con aplicaciones distribuidas en un entorno Cliente/Servidor es necesario para coordinar eventos que ocurran entre ellos. Lo común es que los OS aborden el problema de sincronizar los relojes de cada computadora por medio de un agente de *software* el cual consulta la hora de la computadora y la ajusta abrupta o gradualmente, dependiendo de su configuración, para que esté sincronizado con la del Servidor de la red. Para realizar este servicio como cualquier otro es necesario que las aplicaciones como los emplazamientos de red se comuniquen mediante una semántica de emisión/recepción de mensajes, lo cual es muy útil en situaciones en las que no se desea que clientes y servidores estén muy sincronizados.

El término de “igual a igual” indica que las dos partes de un vínculo de comunicación usan la misma interfaz de protocolos para entablar una conversación en red. Cualquier computadora puede iniciar una conversación con cualquier otra. El protocolo es simétrico y en ocasiones se llama de programa a programa. La interfaz de igual a igual no oculta por completo la red subyacente a un programador de aplicaciones. Uno de los principales protocolos de igual a igual actualmente son las Llamadas a Procedimientos Remotos o RPC (*Remote Procedure Call*).

Las RPC funcionan cuando un proceso Cliente llama a una función en un Servidor remoto y se detiene hasta que recibe los resultados. Los parámetros se transmiten como en cualquier procedimiento ordinario y el proceso es sincrónico. El proceso que emite la llamada espera hasta que obtiene los resultados. En forma oculta, el *software* de RPC de tiempo de ejecución reúne los valores para los parámetros, forma un mensaje y lo envía al Servidor remoto. El Servidor recibe la solicitud, desempaqueta los parámetros, llama al procedimiento y envía la respuesta al Cliente.

Existen entonces diferentes OS muy utilizados que cumplen algunas o todas las características anteriores y que nos permiten trabajar seguros y cómodos (en la medida de lo posible) dentro de una red de computadoras. Actualmente Microsoft Windows y los basados en los estándares POSIX (compatibles o basados en UNIX) son los más utilizados por los usuarios en el hogar o empresariales. El primero es el sistema operativo más utilizado alrededor del mundo puesto que proporciona una gran facilidad en su uso para el usuario final, casi intuitivo y, que permite crear redes de computadoras homogéneas de manera muy fácil, administrar entornos corporativos a través de las redes además de funcionar como base operativa para los servidores de bases de datos y de servicios de Internet de manera eficiente. UNIX es un sistema operativo creado por los Laboratorios Bell en AT&T y esta basado principalmente en el lenguaje de programación C, aunque en sus primeras versiones era ejecutado en computadoras experimentales fue el lenguaje Ensamblador el que le dio forma. Su última versión original fue la VII de 1979. Sin embargo, desde la versión VIII, UNIX fue nada más distribuido dentro de universidades de los Estados Unidos en donde se modificaron muchos segmentos o módulos que lo conformaban hasta que AT&T volvió a retomar el control de esta aplicación al combinar las versiones de las universidades hasta dar forma al UNIX System V. A la par durante todo

este proceso, la creación de un conjunto de normas denominadas POSIX (*Portable Operating System based on UnIX*) o Sistemas Operativos basados en UNIX y las iniciativas de creación de *software* libre o con licencia GNU (*GNU is Not UNIX*), propiciaron la creación de diferentes clones o sistemas similares a UNIX. Entre estos sistemas POSIX se encuentra a MacOS puesto que se deriva de la versión de UNIX creada por la Universidad de Berkeley y el producto de la unión del *software* GNU con el sistema operativo desarrollado por Linus Torvalds: Linux. Las herramientas GNU y el sistema Linux fueron un incentivo para que varias personas u organizaciones de usuarios combinaran ciertas aplicaciones propias junto con el *kernel* y englobarlas en una distribución, de las cuales una de las más famosas es la que creó la compañía Red Hat Software Incorporated.

Internet es la red más grande de computadoras que existe. Esta red nació dentro del sistema operativo UNIX por su fiabilidad, rendimiento y estabilidad. Debido a ello, el conjunto de protocolos usado para transmitir y recibir información a través de ella es el TCP/IP que nació dentro del mundo UNIX y que se ha estandarizado por ser muy flexible para las comunicaciones en red. Debido a lo anterior y a la fuerte influencia que tuvo y tiene UNIX en los sistemas operativos actuales, esta pila de protocolos es la que se utiliza en la mayoría de los sistemas operativos utilizados en las redes de computadoras de las organizaciones.

II.1.2. Aplicaciones Cliente/Servidor.

La tecnología Cliente/Servidor es una forma distinta de entender el mundo de la computación. En este tipo de aplicaciones, los sistemas centralizados ahora son sistemas Servidores que satisfacen las peticiones generadas por los sistemas Clientes. Esta forma da la capacidad a los usuarios de utilizar cualquier tipo de computadora y de Sistema Operativo para trabajar en conjunto con otras máquinas dentro de una red de comunicaciones de igual a igual. Vea Figura 2.3.

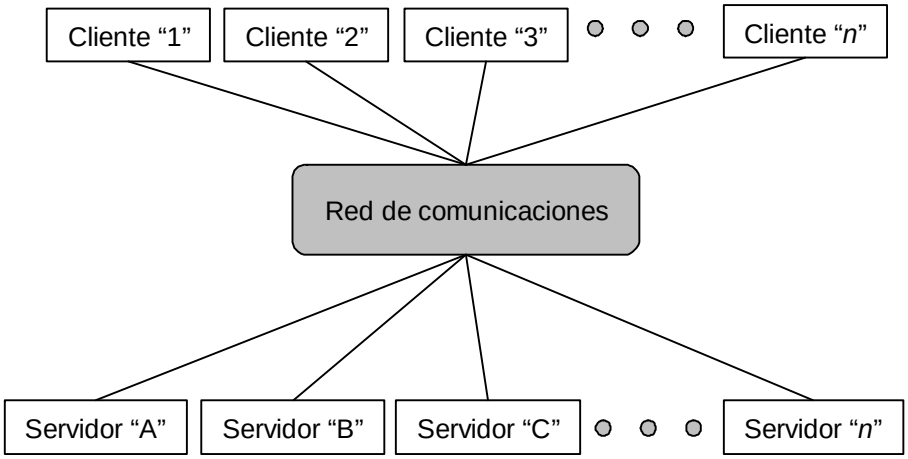

Figura 2.3. Estructura general de un sistema Cliente/Servidor.

El Cliente proporciona la Interfaz Gráfica de Usuario o GUI (*Graphical User Interface*), mientras que el Servidor proporciona acceso a recursos compartidos como, por ejemplo una base de datos. El Cliente como el Servidor son los bloques básicos de construcción de un Sistema Distribuido.

La tecnología Cliente/Servidor engloba entonces los elementos siguientes: una red de computadoras, un protocolo de comunicación, varias computadoras y los sistemas operativos de red correspondientes a cada una de ellas; todos ellos son elementos que deben ser correctamente elegidos para no invertir recursos en unir piezas que sean incompatibles entre sí. Cliente/Servidor se refiere a una arquitectura o división de responsabilidades en las aplicaciones distribuidas de *software*. (Pressman, 2002).

Dentro de la arquitectura Cliente/Servidor, los Clientes y los Servidores son elementos lógicos independientes que operan en conjunto a través de una red de computadoras para realizar una tarea en especial. Los sistemas de este tipo deben poseer las siguientes características:

- Servicio: Es una relación de procesos ejecutados en computadoras o emplazamientos distintos dentro de una red. El Servidor proporciona servicios al Cliente.
- Recursos compartidos: Un Servidor puede atender a muchos clientes al mismo tiempo aunque quieran utilizar el mismo recurso regulando su acceso.

- Protocolos asimétricos: Entre los clientes y servidores se debe de tener una relación de muchos a uno. Son siempre los clientes los que inician el diálogo para solicitar un servicio. Los servidores aguardan pasivamente las solicitudes de los clientes.
- Transparencia de ubicación: Por la definición de un sistema distribuido, el Servidor de un proceso puede incluso estar dentro de la misma computadora (concepto denominado *standalone*) en donde se encuentra el Cliente o en cualquier otra computadora por lo que se tienen que ocultar o hacer transparente la ubicación de estos elementos lógicos que componen a Cliente/Servidor mediante un redireccionamiento de las llamadas de servicio en caso necesario. Un programa puede ser un Cliente, un Servidor o ambos.
- Mezcla e igualdad: Las aplicaciones Cliente/Servidor idealmente son independientes de la plataforma de *hardware* o de *software* en la que se desarrolle.
- Intercambios basados en mensajes: Clientes y servidores son sistemas acoplados que interactúan entre sí por medio de protocolos y de mecanismos de transmisión de mensajes. El mensaje es el mecanismo de entrega de solicitudes y respuestas de un servicio.
- Encapsulamiento de servicios: Encapsular en términos de Programación Orientada a Objetos es una característica que permite ver a un objeto como una caja negra en la que se ha introducido de alguna manera toda la información relacionada con dicho objeto. Permite entonces la manipulación de los objetos como unidades básicas permaneciendo oculta su estructura interna (Ceballos, 2000). Si hacemos una analogía, esta característica nos permite que varios servidores nos puedan proporcionar un servicio porque la interfaz de datos de entrada en un mensaje no cambia.
- Facilidad de escalabilidad: Los sistemas Cliente/Servidor pueden escalarse horizontal o verticalmente. La forma horizontal significa la adición o eliminación de estaciones de trabajo del tipo Cliente. La escalabilidad vertical significa emigrar a un Servidor a otra (s) computadora (s) más avanzada (s) y con mayores recursos.

- Integridad: El código del Servidor y los datos del mismo se conservan centralmente lo que resulta en un mantenimiento de menor costo y la protección de la integridad de los datos compartidos. Los clientes mantienen su individualidad e independencia de igual manera.

Para desarrollar aplicaciones distribuidas basándose en los sistemas Cliente/Servidor se necesita el conocimiento de ciertas áreas de interés, como:

- La comunicación en red de computadoras.
- El diseño de bases de datos distribuidas.
- El conocimiento en diseño de interfaces gráficas de usuario.

Cliente/Servidor es un tipo especial de Sistemas Distribuidos. El Cliente es la aplicación (a quién se le conoce como parte frontal o *front – end*) y el Servidor (*back – end*) que puede ser de diferentes tipos dependiendo del área de interés del constructor del sistema distribuido. Entre estos tipos tenemos a:

- Servidores de archivos.
- Servidores de *software* de grupo.
- Servidores Web.
- Servidores de correo electrónico.
- Servidores de impresión.
- Servidores de aplicaciones.
- Servidores de bases de datos.

Los servidores de bases de datos son elementos que almacenan grandes colecciones de datos estructurados. En un entorno de bases de datos Cliente/Servidor los clientes envían consultas a la base de datos, normalmente utilizando alguna GUI. Estas

consultas se envían al Servidor a través del "Lenguaje de Base de Datos SQL[5]". El Servidor de datos lee el código SQL, lo interpreta y, a continuación, lo visualiza en algún objeto de la misma GUI, por ejemplo, una caja de texto. El punto clave aquí es que el Servidor de bases de datos lleva a cabo todo el procesamiento, donde el Cliente lleva a cabo los procesos de extraer una consulta de algún objeto de entrada, tal como un campo de texto, enviar la consulta y visualizar la respuesta del Servidor de bases de datos en algún objeto de salida tal como un cuadro de desplazamiento (Elmasri y Navathe, 2000).

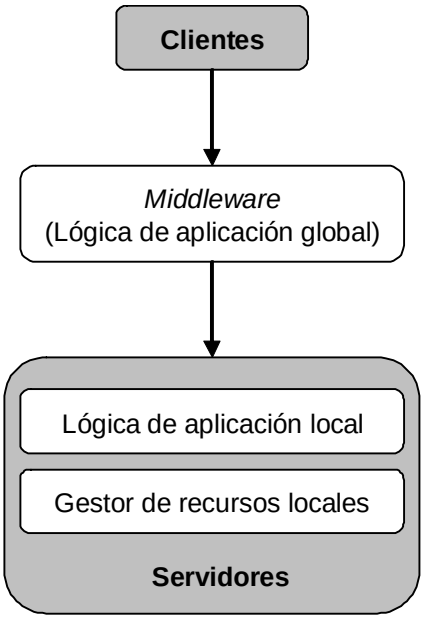

Figura 2.4. El *middleware* dentro de Cliente/Servidor.

Además de los componentes principales (Cliente y Servidor), existe otro bloque de construcción del *software* que suele denominarse *Software* Intermedio o *Middleware* en todos los sistemas de Cliente/Servidor. El *software* intermedio es definido como «el sistema nervioso de un sistema Cliente/Servidor» (Orfali, et. al. 1998). Establece la infraestructura que hace posible que los componentes Cliente y Servidor interactúen entre sí. Vea Figura 2.4. El *middleware* es un término que abarca todo el *software* distribuido necesario para el

[5] No son las siglas de "Lenguaje de Consultas Estructurado". Lo anterior era cierto para los prototipos originales, pero no cuando se hizo estándar. Las siglas SQL no significan nada, no son una abreviatura. Simplemente se le debe conocer como "Lenguaje de base de datos SQL" (Melton y Eisenberg, 2002).

soporte de interacciones entre clientes y servidores. Es el *software* que ocupa la parte intermedia del sistema de Cliente/Servidor y es el enlace que permite que un Cliente obtenga un servicio de un Servidor. Este elemento inicia en un módulo de comunicaciones del Cliente que se emplea para invocar un servicio y comprende la transmisión de la solicitud por la red y la respuesta resultante; no incluye el *software* que presta el servicio real puesto que eso corresponde al Servidor, ni la interfaz de usuario y la lógica de la aplicación que es parte de los dominios del Cliente.

El *middleware* se divide en dos grandes clases:

- El de servicios generales es el que está asociado a los servicios generales que requieren todos los clientes y servidores. El *software* típico que se utiliza como tal es: el *software* para llevar a cabo las comunicaciones utilizando el conjunto de protocolos TCP/IP u otros protocolos de red; el del sistema operativo que, por ejemplo, mantiene un almacenamiento distribuido de archivos. Este es una colección de archivos que se esparcen por un sistema distribuido. El propósito de esta parte del sistema operativo es asegurar que los usuarios pueden acceder a estos archivos de forma que no necesiten conocer su localización, el *software* de autenticación que comprueba que un usuario válido que desee utilizar un sistema distribuido pueda hacerlo, el *software* intermedio orientado a mensajes que gestionan el envío de mensajes desde clientes a servidores y viceversa, por último, los componentes de la aplicación diseñados especialmente para el sistema distribuido.
- El de servicios específicos es el *software* asociado a un servicio en particular. Entre los ejemplos típicos de este tipo de *software* se incluyen al *software* que permite a bases de datos diferentes conectarse a una red Cliente/Servidor. Probablemente el mejor ejemplo de este tipo de *software* sea el de Conectividad Abierta de Bases de Datos u ODBC (*Open Data Base Connectivity*) producida por Microsoft. Este permite que existan en una red la vasta mayoría de sistemas de gestión de bases de datos; *software* específico de objetos distribuidos, como es CORBA. CORBA es una tecnología objetos distribuida que permite a objetos escritos en una gran variedad de lenguajes

que coexistan en una red de tal forma que cualquier objeto pueda enviar mensajes a otro objeto. El *software* intermedio de CORBA tiene que ver con funciones tales como configurar objetos distribuidos, comunicación y seguridad entre ellos; *software* intermedio de Web asociado al protocolo HTTP; *software* intermedio de aplicaciones de grupo que administra los archivos que describen a los equipos de trabajo y sus interacciones; y al *software* intermedio asociado a productos de seguridad específicos. Un buen ejemplo es el *software* intermedio asociado a lo que se conoce como Conexiones Seguras (*Secure Sockets*). Estas son conexiones que se pueden utilizar para enviar datos seguros en una red de tal forma que hace imposible que intrusos escuchen a escondidas los datos.

Al igual que los otros dos componentes principales, el *middleware* también cuenta con un Administrador de Sistemas Distribuidos o DSM (*Distribuited System Management*) para mantener unido a todo el sistema.

Las aplicaciones Cliente/Servidor también se pueden clasificar en la forma en la que la aplicación distribuida se divide entre el Cliente y el Servidor; decisión que recae en el ingeniero de *software* que las construye. Cuando la mayor parte de la funcionalidad asociada a cada uno de los tres componentes se le asocia al Servidor, se ha creado un diseño de Servidor Pesado (*Fat Server*). A la inversa, cuando el Cliente implementa la mayor parte de los componentes de la interacción/presentación con el usuario, de aplicación y de base de datos, se tiene un diseño de Cliente Pesado (*Fat Client*); lo óptimo es encontrar un balance adecuado de las operaciones y servicios de las aplicaciones Cliente/Servidor; situación que se puede apreciar en la Figura 2.5.

Los clientes pesados suelen encontrarse cuando se implementan arquitecturas de Servidor de archivo y Servidor de bases de datos. En este caso, el Servidor proporciona apoyo de gestión de datos, pero todo el *software* de aplicación y de GUI reside en el Cliente. Los servidores pesados son los que suelen diseñarse cuando se implementan sistemas de transacciones y de trabajo en grupo. El Servidor proporciona el apoyo de la aplicación necesario para responder a transacciones y comunicaciones que provengan de los clientes. El *software* de Cliente se centra en la gestión de la GUI y de comunicación.

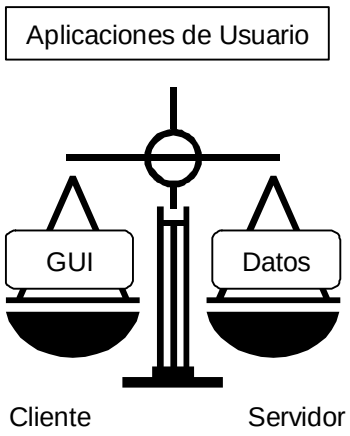

Figura 2.5. Diseño óptimo de aplicaciones Cliente/Servidor.

Un Cliente pesado implementa la mayoría de las aplicaciones específicas de la aplicación en el lado Cliente. Un Cliente ligero (*Thin Client*) relega la mayor parte del proceso en el lado Servidor. Se puede utilizar clientes y servidores pesados para ilustrar el enfoque general de asignación de componentes de *software* Cliente/Servidor, sin embargo, un enfoque más granular para la asignación de componentes de *software* define cinco configuraciones diferentes:

- Presentación distribuida. En este enfoque Cliente/Servidor rudimentario, la lógica de la base de datos y la lógica de la aplicación permanecen en el Servidor, típicamente en una computadora central. El Servidor contiene también la lógica de preparar información de pantalla utilizando un *software* especial para transformar ésta en una presentación GUI en una computadora.
- Presentación remota. En esta extensión del enfoque de presentación distribuida, la lógica primaria de la base de datos y de la aplicación permanece en el Servidor, y los datos enviados por él serán utilizados por el Cliente para preparar la presentación del usuario.
- Lógica distribuida. Se asignan al Cliente todas las tareas de presentación del usuario y también los procesos asociados a la introducción de datos tales como la validación de nivel de campo, la formulación de consultas al Servidor

y las solicitudes de informaciones de actualizaciones del Servidor. Se asignan al Servidor las tareas de gestión de las bases de datos, y los procesos para las consultas del Cliente, para actualizaciones de archivos del Servidor, para control de versión de clientes y para aplicaciones de ámbito general de la empresa.

- Gestión de datos remota. Las aplicaciones del Servidor crean una nueva fuente de datos dando formato a los que se han extraído de algún otro lugar (por ejemplo, de una fuente de nivel corporativo). Las aplicaciones asignadas al Cliente se utilizan para explotar los nuevos datos a los que ha dado formato mediante el Servidor. En esta categoría se incluyen a los Sistemas de Apoyo a la toma de Decisiones.
- Bases de datos distribuidas. Los datos de que consta la base de datos se distribuyen entre múltiples clientes y servidores. El Cliente debe de admitir componentes de *software* de gestión de datos así como componentes de aplicación y de GUI.

Actualmente se ha dado mucha importancia a la tecnología de Cliente ligero. Un Cliente ligero es la llamada Computadora de Redes o NC (*Network Computer*) que relega todo el procesamiento de la aplicación a un Servidor pesado. Los clientes ligeros (computadoras de red) ofrecen un costo por unidad sustancialmente más bajo a una pérdida de rendimiento pequeña o nada significativa en comparación con las computadoras de escritorio.

El enfoque sobre como dividir a los sistemas Cliente/Servidor es denominado como “Arquitecturas de *n* planos”. Todo se reduce al modo en el que la aplicación distribuida se divide en unidades funcionales que más tarde puedan asignarse ya sea al Cliente o a uno o más servidores. Las unidades funcionales más comunes son la interfaz de usuario, la lógica de los negocios y los datos compartidos:

- Componente de interacción con el usuario y presentación. Este componente implementa todas las funciones que típicamente se asocian a una GUI.

- Componente de aplicación o lógica de negocio. Este componente implementa los requisitos definidos en el contexto del dominio en el cual funciona la aplicación. Por ejemplo, una aplicación de negocios podría producir toda una gama de informes impresos basados en entradas numéricas, cálculos, información de una base de datos y otros aspectos. Una aplicación de *software* de grupo podría proporcionar funciones para hacer posible la comunicación mediante boletines de anuncios electrónicos o de correo electrónico, y en nuestro caso de estudio esto conllevaría la preparación de informes como los que describen la venta de libros. En ambos casos, el *software* de la aplicación se puede descomponer de tal modo que alguno de los componentes resida en el Cliente y otros residan en el Servidor.
- Componente de gestión de bases de datos. Es un componente que lleva a cabo la manipulación y gestión de datos por una aplicación. La manipulación y gestión de datos puede ser tan simples como la transferencia de un registro, o tan compleja como el procesamiento de sofisticadas transacciones SQL.

Aun cuando no existan reglas absolutas que describan la distribución de componentes de aplicaciones entre el Cliente y el Servidor, suelen seguirse las siguientes líneas generales:

- El componente de presentación/interacción suele ubicarse en el Cliente. La disponibilidad de entornos basados en ventanas y de la potencia de cómputo necesaria para una interfaz gráfica de usuario hace que este enfoque sea eficiente en términos de costo.
- Si es necesario compartir la base de datos entre múltiples usuarios conectados a través de la red LAN, entonces la base de datos suele ubicarse en el Servidor. El sistema de gestión de la base de datos y la capacidad de acceso a la base de datos, se asignan al Servidor junto con la base de datos física.
- Los datos estáticos que se utilicen deberían de asignarse al Cliente. Esto sitúa los datos más próximos al usuario que tiene necesidad de ellos y minimiza un tráfico de red innecesario y la carga del Servidor. El resto de los componentes

de la aplicación se distribuye entre el Cliente y Servidor basándose en la distribución que optimice las configuraciones de Cliente y Servidor y de la red que los conecta.

La decisión final acerca de la distribución de componentes debería estar basada no solamente en la aplicación individual, sino en la mezcla de aplicaciones que esté funcionando en el sistema. Habría que tener en cuenta que a medida que madura el uso de la arquitectura Cliente/Servidor, la tendencia es ubicar la lógica de la aplicación en el Servidor. Esto simplifica la implantación de actualizaciones de *software* cuando se hacen cambios en la lógica de la aplicación.

Una arquitectura de dos capas de una aplicación Cliente/Servidor consiste en una capa de lógica y presentación, y otra capa de bases de datos. La primera tiene que ver con presentar al usuario varios conjuntos de objetos visuales y llevar a cabo el procesamiento que requieren los datos producidos por el usuario y los devueltos por el Servidor. Por ejemplo, esta capa contendría el código que monitoriza las acciones de pulsar botones, el envío de datos al Servidor, y cualquier cálculo local necesario para la aplicación. Estos datos se pueden almacenar en una base de datos convencional, en un archivo simple o pueden ser incluso los datos que están en la memoria. Esta capa reside en el Servidor. Normalmente las arquitecturas de dos capas se utilizan cuando se requiere mucho procesamiento de datos.

Por ejemplo, la arquitectura de construcción de un Servidor Web. Esto es porque en el navegador de Web del Cliente reside la capa lógica y de presentación mientras que los datos del Servidor Web residen en las capas de la base de datos; vea Figura 2.6. Sin embargo, esta arquitectura está altamente orientada a sucesos específicos y no a los datos subyacentes de una aplicación; de aquí que la reutilización sea algo complicado. Lo anterior es resuelto en gran medida con la arquitectura de tres capas; vea Figura 2.7. Ésta se compone de una capa de presentación, una capa de procesamiento (o capa de Servidor de solicitudes) y una capa de bases de datos.

La capa de presentación es la responsable de la presentación visual de la aplicación. La capa de base de datos contiene los datos de la aplicación y la capa de procesamiento es la responsable del procesamiento que tiene lugar en la aplicación. Por

ejemplo, en una transacción bancaria el código de la capa de presentación se relacionaría simplemente con la monitorización de un suceso y con el envío de datos a la capa de procesamiento. Esta capa intermedia contendría los objetos que corresponden con las entidades de la aplicación, por ejemplo, en una aplicación bancaria los objetos típicos serían los bancos, el Cliente, las cuentas y las transacciones.

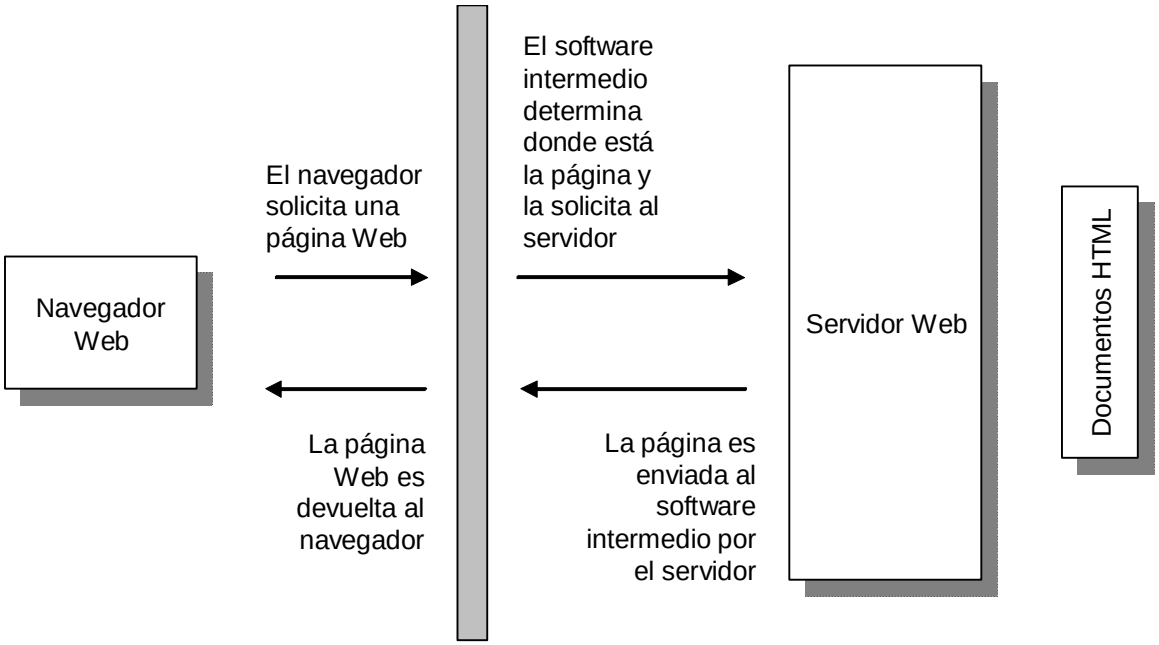

Figura 2.6. Arquitectura de un Servidor Web.

La capa final sería la capa de la base de datos. Ésta estaría compuesta de los archivos que contienen los datos de la aplicación. La capa intermedia es la que conlleva a capacidad de mantenimiento y de reutilización. Contendrá objetos definidos por clases reutilizables que se pueden utilizar una y otra vez en otras aplicaciones. Estos objetos se suelen llamar objetos de negocios y son los que contiene la gama normal de constructores, métodos para establecer y obtener variables, métodos que llevan a cabo cálculos y métodos, normalmente privados, en comunicación con la capa de bases de datos. La capa de presentación enviará mensajes a los objetos de esta capa intermedia, la cual o bien respondería entonces directamente o mantendrá un diálogo con la capa de la base de datos que se mandarían como respuestas a la capa de presentación.

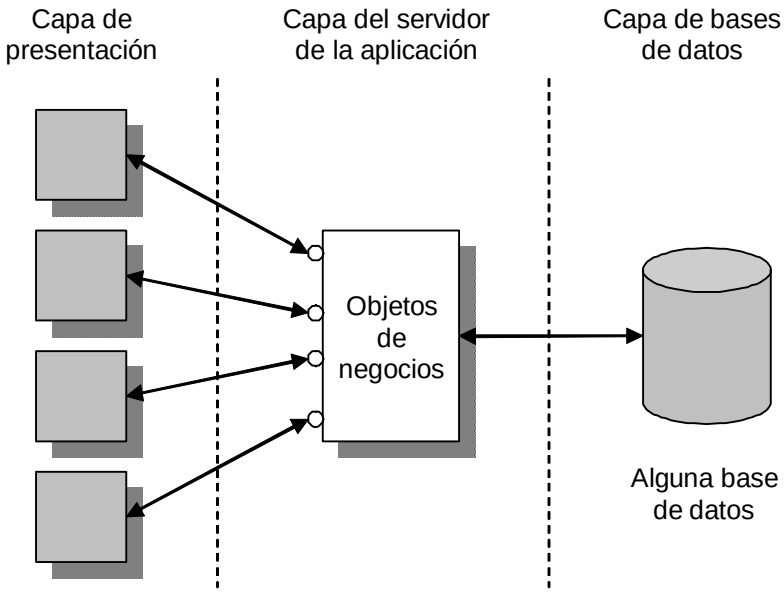

Figura 2.7. Una arquitectura de tres capas.

El lugar donde va a residir la capa intermedia depende de la decisión sobre el diseño. Podría residir en el Servidor, que contiene la capa de base de datos; por otro lado, podría residir en un Servidor separado. La decisión sobre dónde colocar esta capa dependerá de las decisiones obre diseño que se apliquen, dependiendo de factores tales como la cantidad de carga que tiene un Servidor en particular y la distancia a la que se encuentra el Servidor de los clientes. La localización de esta capa no le resta valor a las ventajas que proporciona una arquitectura de tres capas:

- En primer lugar, la arquitectura de tres capas permite aislar la tecnología que implementa la base de datos de forma que sea fácil cambiar esta tecnología. Por ejemplo, una aplicación podría utilizar en primer lugar la tecnología relacional para la capa de base de datos; si una base de datos basada en objetos funciona tan eficientemente como la tecnología relacional que se pudiera entonces integrar fácilmente todo lo que se necesitaría cambiar serían los métodos para comunicarse con la base de datos.

- Utiliza mucho código lejos del Cliente. Los clientes que contienen mucho código se conocen como clientes pesados. En un entorno en donde se suele necesitar un cambio, estos clientes pesados pueden convertirse en una pesadilla de mantenimiento. Cada vez que se requiere un cambio la organización tiene que asegurarse de que se ha descargado el código correcto a cada Cliente. Con una capa intermedia una gran parte del código de una aplicación Cliente/Servidor reside en un lugar (o en un número reducido pequeño de lugares si se utilizan servidores de copias de seguridad) y los cambios de mantenimiento ocurren de forma centralizada.

La idea de las tres capas encaja con las prácticas de hoy en día: todo el procesamiento tiene lugar por medio de los mensajes que se envían a los objetos y no mediante trozos de código asociados a cada objeto en la capa de presentación que se está ejecutando. Independientemente de la estructura o la arquitectura a utilizar en la construcción de un sistema Cliente/Servidor y del *middleware*, éstos utilizan todos los recursos del sistema en general para funcionar y poder comunicarse entre sí (Pressman, 2002).

El Sistema Operativo de Red o NOS (*Network Operating System*) es un elemento fundamental en el sistema Cliente/Servidor. Los programas servidores son objeto de un alto nivel de concurrencia y lo ideal sería que a cada uno de los clientes a los que el Servidor debe soportar concurrentemente se le asignara una tarea independiente. Un NOS multitarea es especialmente apto para la administración de tareas porque es el método natural para simplificar la codificación de aplicaciones complejas que pueden dividirse en una serie de tareas concurrentes y lógicamente distintas. Su intervención eleva el desempeño y también la existencia de mecanismos para la coordinación de tareas y los intercambios de información. Los servidores requieren asimismo un alto nivel de concurrencia dentro de un solo programa. El código del Servidor correrá más eficientemente si las tareas se distribuyen entre las diversas partes del mismo programa, en lugar de destinarse a programas distintos. Cuando las tareas se asignan al mismo programa, su creación es más rápida y es más fácil su acceso a la información compartida. La preferencia, la prioridad de tareas y los hilos de concurrencia ayudan mucho. Sin embargo para manejar la

sincronización de las tareas, los semáforos ayudan a que el Servidor atienda peticiones que requieran del cumplimiento de una tarea anterior o posterior a través de la comunicación entre procesos.

Los servidores también necesitan de los NOS para comunicarse con el mayor número de computadoras y plataformas operativas de los clientes (especialmente en un entorno heterogéneo de plataformas) utilizando los archivos o los recursos que estén compartidas dentro de cada una de esos clientes y para ubicarlos dentro de la red se necesita de los NOS para localizar servidores y recursos en red a través de los nombres de los recursos en general para que el Servidor sea capaz de registrarlos de manera dinámica. Además los sistemas de autenticación y autorización le deben de señalar al Servidor que el usuario que accede es quién dice ser, puesto que el sistema de autorización determina si un Cliente autenticado tiene permiso para utilizar un servicio remoto.

Por el lado del Cliente, todas las aplicaciones precisan un mecanismo para comunicar solicitudes de servicio y de archivos a un Servidor. Los clientes funcionan mejor dentro de un robusto entorno multitarea. Es importante utilizar un NOS capaz de proteger los programas de colisiones o impactos dentro de la memoria. Ningún programa Cliente debe de provocar la descompostura del sistema. En general, se necesita de un mecanismo de solicitud/respuesta con transparencia local/remota, un sistema para la transferencia de archivos, priorizar las tareas, comunicaciones entre procesos y GUI fáciles de utilizar y comprender.

La actividad de modelado de requisitos para los sistemas Cliente/Servidor difiere poco de los métodos de modelado de análisis que se aplicaban para la arquitectura de computadoras más convencionales. Consiguientemente los principios básicos de análisis y los métodos de modelado de los mismos son igualmente aplicables al *software* Cliente/Servidor, es decir, a nivel del análisis hay muy poca diferencia entre un sistema distribuido y un sistema local, y esto se basa en que el modelo de análisis de un sistema no contendrá ningún dato de diseño como el hecho de que tres computadoras y no una, están llevando a cabo algún procesamiento. Se deberá destacar, sin embargo, que dado que muchos sistemas Cliente/Servidor modernos hacen uso de componentes reutilizables, también se aplican las actividades de calificación asociadas a la Ingeniería de *Software* Basada en Componentes.

Dado que el modelado de análisis evita la especificación de detalles de implementación, solo cuando se haga la transición al diseño se considerarán los problemas asociados a la asignación de *software* al Cliente y al Servidor, a menos que el modelo de desarrollo de *software* posibilite la iteración del proceso.

El proceso de diseño implica transformar el modelo de análisis en algún modelo físico que se implementa en los elementos de *hardware* del sistema. Es necesario que el constructor de sistemas distribuidos conozca una serie de principios de diseño basándose en las necesidades o requisitos del sistema mismo así como al tipo de Servidor a utilizar. A continuación se presenta una serie de recomendaciones de diseño (Pressman, 2002):

- Correspondencia del volumen de transmisión con los medios de transmisión. Esto significa que para un tráfico denso de datos en un sistema distribuido se deberían utilizar medios de transmisión rápidos (y caros). El proceso de asignar tales medios normalmente tiene lugar después de hacer tomado decisiones sobre la potencia de procesamiento de distribución en un sistema y, algunas veces, conlleva unas ligeras iteraciones al final de la fase de diseño.
- Mantenimiento de los datos más usados en un almacenamiento rápido. Requiere que el diseñador examine los patrones de datos en un sistema y asegure que los datos a los que se accede frecuentemente se guarden en algún medio de almacenamiento rápido. En muchos sistemas tales de datos pueden constituir no más del 5% de los datos originales almacenados en el sistema, y de esta manera permite utilizar con frecuencia las estrategias que conllevan el almacenamiento de estos datos dentro de la memoria principal.
- Mantenimiento de los datos cerca de donde se utilizan. Este principio de diseño intenta reducir el tiempo que pasan los datos por medios lentos de transmisión. Muchos de estos sistemas son en donde los usuarios acceden con frecuencia a un subconjunto de datos. Por ejemplo, un sistema distribuido usado en una aplicación bancaria contendría bases de datos con datos de las cuentas de los clientes, en donde la mayor parte de las consultas a las bases de datos de las sucursales las realizarán los clientes de esa sucursal; entonces, si los datos de un sistema bancario se distribuyen a los servidores de las

sucursales, y los datos asociados a los clientes de esa sucursal están en esa sucursal, el resto de los datos podrían estar en otros servidores con otras ubicaciones, y cualquier consulta que se originara sobre los datos se tendría que comunicar a través de las líneas de transmisión.

- Permitir la duplicación de datos. La duplicación consiste en mantener múltiples copias de datos en un sistema al mismo tiempo. Existen muchas razones para la duplicación de datos. La primera es que hay que asegurar la redundancia que permite que un sistema distribuido continúe funcionando aún cuando una computadora con datos importantes quede fuera de servicio normalmente por un mal funcionamiento del *hardware*. La otra razón es que proporciona la manera de permitir el mantenimiento de los datos cerca de donde se utilizan. Esto implica que las bases de datos utilizadas de esta forma tengan una relación dinámica para que se mantengan informadas en el caso de la existencia de una actualización a la base de datos y que conserven la misma información, es decir, que cualquier cambio efectuado a una base de datos debe estar reflejado en sus múltiples copias. Esto también implica la aparición de retrasos porque las transacciones permanecen en cola esperando a que la base de datos se sincronice con otras bases de datos. Esto no significa que se tenga que utilizar la duplicación de datos, lo que significa es que se necesita un diseño cuidadoso para minimizar la cantidad de gastos asociados a él en las transmisiones.
- Eliminar cuellos de botella. En un sistema distribuido, un Servidor tiene con frecuencia un cuello de botella; tiene que manipular tanto tráfico que se construyen grandes colas de transacciones esperando a ejecutarse, con el resultado de que los servidores están esperando los resultados del procesamiento estarán, en el mejor de los casos, ligeramente cargados y, en el peor, inactivos. La estrategia normal para manipular cuellos de botella es compartir la caga de procesamiento entre los servidores, normalmente servidores físicamente cerca del que está sobrecargado.
- La minimización de la necesidad de un gran conocimiento del sistema. Los sistemas distribuidos suelen necesitar conocer el estado del sistema completo,

por ejemplo, podría ser que necesitaran conocer la cantidad de registros de una base de datos central. El hecho de necesitar este conocimiento genera más tráfico reduciendo así la eficiencia de un sistema, ya que generará tráfico extra a lo largo de las líneas de transmisión. El diseñador de un sistema distribuido en primer lugar necesita minimizar que el sistema dependa de datos globales, y entonces asegurar que el conocimiento necesario se comunique rápidamente a aquellos componentes del sistema que lo requieran.

- La agrupación de datos afines en la misma ubicación. Los datos relacionados deberían de estar dentro del mismo Servidor. El ubicar por separado los datos en diferentes servidores asegura que los medios de baja transmisión y muy cargados se cargarán incluso más. El diseñador de un sistema distribuido debe asegurarse de que los datos relacionados gracias al hecho de que se suelen recuperar juntos tendrán que residir lo más cerca posible, preferiblemente en el mismo Servidor, o si no, y no de manera tan preferible, en servidores conectados a través de medios de transmisión rápidos tales como los medios utilizados en una red de área local.
- El considerar la utilización de servidores de dedicados a funciones frecuentes. Algunas veces se puede lograr un mayor rendimiento mediante la utilización de un Servidor de empleo específico para una función en particular en lugar de, por ejemplo, un Servidor de bases de datos.
- Correspondencia de la tecnología con las exigencias de rendimiento. Muchas de las tecnologías utilizadas en las redes tienen pros y contras, y un factor importante aquí son las demandas de rendimiento de una tecnología en particular. Por ejemplo, como medio de comunicación, las conexiones (*sockets*) normalmente son un medio de comunicación mucho más rápido que los objetos distribuidos. Cuando el diseñador elige la tecnología debe de tener conocimiento de la transmisión y de las cargas de procesamiento que conlleva, y seleccionar una tecnología que minimice estas cargas.
- Empleo del paralelismo. Una de las ventajas principales de la tecnología Cliente/Servidor es el hecho de que se pueden añadir servidores y, hasta cierto punto, elevar el rendimiento del sistema. Muchas funciones del comercio

electrónico pueden beneficiarse de la ejecución que están llevando a cado diferentes servidores en paralelo. Esta no es una decisión sencilla. Mediante el empleo de varios servidores, el diseñador está creando la necesidad de que estos servidores se comuniquen, por ejemplo, un Servidor puede que necesite a otro para completar una tarea en particular antes de finalizar la suya propia. Esta comunicación puede introducir retrasos y, si el diseñador no tiene cuidado, pueden negar los avances del rendimiento que se han logrado utilizando el paralelismo.

- Utilización de la compresión de datos. Un grupo de algoritmos que compriman datos y que reduzcan el tiempo de transferencia entre un componente de un sistema distribuido y otro componente, es necesario en un buen sistema distribuido. El único gasto que se requiere para utilizar esta técnica es el tiempo del procesador y la memoria necesaria para llevar a cabo la compresión en la computadora del emisor y la descompresión en la computadora del receptor.
- Diseño para el fallo. Un fallo del *hardware* en la mayoría de los sistemas distribuidos es una catástrofe. Una parte importante del proceso de diseño es analizar los fallos que podrían aparecer en un sistema distribuido y diseñar el sistema con suficiente redundancia como para que dicho fallo no afecte seriamente y, en el mejor de los casos, que se pueda reducir el tiempo de respuesta de ciertas transacciones. Una decisión normal que suele tomar el diseñador es duplicar los servidores vitales para el funcionamiento de un sistema distribuido. Una estrategia de los sistemas de alta integridad es que un Servidor que se reproduzca tres veces y que cada Servidor lleve a cabo la misma tarea en paralelo. Cada Servidor produce un resultado a comparar. Si los tres servidores aceptan el resultado, éste pasa a cualquier usuario o Servidor que lo requiera; si uno de los servidores no está de acuerdo, entonces surge un problema y el resultado de la mayoría se pasa informando al administrador de sistemas del posible problema. La duplicación de servidores como estrategia de mitigación de fallos puede utilizarse junto con el diseño de un sistema para lograr el paralelismo en las tareas.

- Minimizar la latencia. Cuando los datos fluyen de una computadora a otra en un sistema distribuido a menudo tiene que atravesar otras computadoras. Algunas de estas computadoras pueden que ya tengan unos datos que expidan funcionalidad; es posible que otras procesen los datos de alguna manera. El tiempo que tardan las computadoras es lo que se conoce como latencia. Un buen diseño de sistema distribuido es el que minimiza el número de computadoras intermediarias.

Por último, la naturaleza distribuida de los sistemas Cliente/Servidor plantea un conjunto de problemas específicos para las pruebas de *software*, dentro de estos están los siguientes:

- Consideraciones del GUI del Cliente.
- Consideraciones del entorno destino y de la diversidad de plataformas.
- Consideraciones de bases de datos distribuidas (incluyendo datos duplicados).
- Entornos de destino que no son robustos.
- Relaciones de rendimiento no lineales.

La estrategia y las tácticas asociadas a la comprobación Cliente/Servidor no se produce en tres niveles diferentes: (1) las aplicaciones de Cliente individuales se prueban de modo desconectado (el funcionamiento del Servidor y de la red subyacente no se consideran); (2) las aplicaciones de *software* del Cliente y del Servidor asociado se prueban al unísono, pero no se ejecutan específicamente las operaciones de red; (3) se prueba la arquitectura completa de Cliente/Servidor, incluyendo el rendimiento y funcionamiento de la red. Aún cuando se efectúen muchas clases distintas de pruebas en cada uno de los niveles de detalle, es frecuente encontrar los siguientes enfoques de pruebas para aplicaciones Cliente/Servidor:

- Pruebas de función de aplicación. Se prueba la funcionalidad de las aplicaciones Cliente. En esencia, la aplicación se prueba en solitario en un intento de descubrir errores de funcionamiento.

- Pruebas de Servidor. Se prueban la coordinación y las funciones de gestión de datos del Servidor. También se considera el rendimiento del Servidor (tiempo de respuesta y transvase de datos en general).
- Pruebas de bases de datos. Se prueba la precisión e integridad de los datos almacenados en el Servidor, se examinan las transacciones enviadas por las aplicaciones Cliente para asegurar que los datos se almacenen, actualicen y recuperen adecuadamente. También se prueba el archivado.
- Pruebas de transacciones. Se crea una serie de pruebas adecuada para comprobar que todas las clases de transacciones se procesen de acuerdo a los requisitos. Las transacciones hacen especial hincapié en la corrección de procesamiento y también en los temas de rendimiento.
- Pruebas de comunicaciones a través de la red. Estas pruebas verifican que la comunicación entre los nudos de la red se produzca correctamente, y que el paso de mensaje, las transacciones y el tráfico de red relacionado tenga lugar sin errores. También se pueden efectuar pruebas de seguridad de la red como parte de esta actividad de prueba.

II.2. Sistemas de Base de Datos Relacional.

Las bases de datos han evolucionado con el paso del tiempo, en ser simplemente una aplicación informática hasta ser una parte importante dentro de un entorno informático en la actualidad. Vivimos en una era de información en donde todos los datos que le dan forma se encuentran almacenados en grandes sistemas y se encuentran repartidas por todo el mundo en forma de bits, además de poder ser utilizadas a cualquier hora del día, gracias a las redes de computadoras.

Una base de datos es un conjunto de datos relacionados entre sí; datos que representan algún aspecto del mundo real, que pueden registrarse de manera lógica y coherente para que tengan un significado implícito y un propósito específico para un grupo de usuarios que la necesite (Elmasri y Navathe, 2000).

Las bases de datos se pueden crear y manipular con un grupo de aplicaciones escritas específicamente para esa tarea denominado Sistema de Administración para Bases de Datos o DBMS (*Data Base Management System*). La Figura 2.8 muestra una imagen simplificada de un Sistema de Administración de Base de Datos.

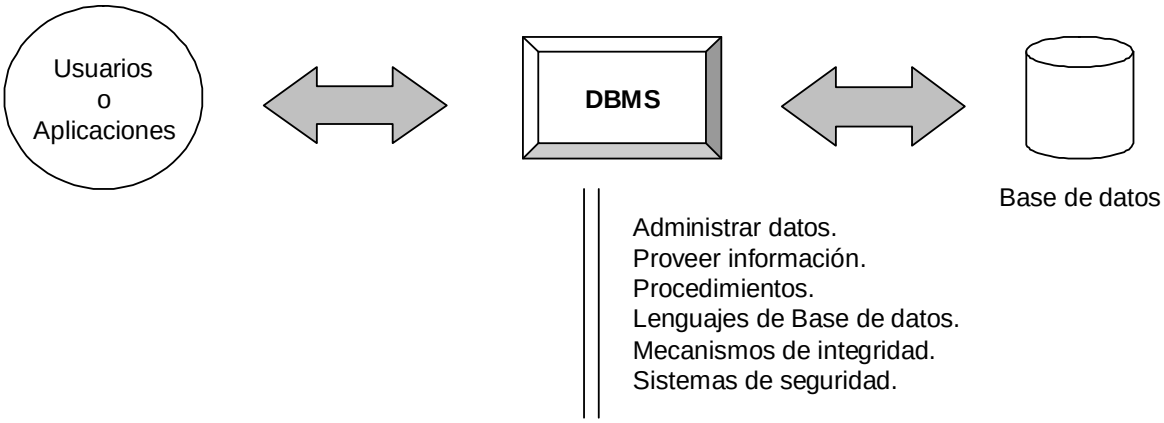

Figura 2.8. Un sistema de administración de bases de datos.

Un DBMS consiste en una base de datos y de un conjunto de aplicaciones para acceder a dichos datos. Su propósito es proporcionar un entorno que sea tanto práctico como eficiente de usar para la recuperación de información y almacenamiento de los datos.

Por lo tanto, un DBMS es un *software* de propósito general (normalmente) que facilita el proceso de definir, construir y manipular bases de datos para diversos usos. Las ventajas de una base de datos computarizada están en el compartir los datos con otros usuarios a los cuales tengan derecho (seguridad en el acceso), en menos redundancia o repetición de los datos para evitar la inconsistencia de los mismos y provocar que sean íntegros, en la velocidad de recuperación de información y actualización de los datos además de tener un control centralizado de los mismos.

Para que el sistema sea útil, debe de recuperar los datos eficientemente. Para esto se han propuesto diseños de estructuras de datos complejas para la representación de los datos en la base de datos. Como muchos usuarios no se encuentran en ocasiones familiarizados en estos temas, los desarrolladores de aplicaciones esconden la complejidad del sistema a través de varios niveles de abstracción para simplificar la interacción; vea Figura 2.9:

- Nivel físico. El nivel más bajo de abstracción describe cómo se almacenan realmente los datos. En el nivel físico se describen en detalle las estructuras de datos complejas de bajo nivel.
- Nivel lógico y conceptual. El siguiente nivel más alto de abstracción describe qué datos se almacenan en la base de datos y qué relaciones existen entre esos datos. La base de datos completa se describe así en términos de un número pequeño de estructuras relativamente simples. Aunque la implementación de estructuras simples en el nivel lógico puede involucrar estructuras complejas del nivel físico, los usuarios del nivel lógico no necesitan preocuparse de esta complejidad. Los Administradores de Bases de Datos o DBA (*Data Base Administrator*), que deben decidir la información que se mantiene en la base de datos, utilizan el nivel lógico de abstracción.
- Nivel de vistas. El nivel más alto de abstracción describe sólo parte de la base de datos completa. A pesar del uso de estructuras más simples en el nivel lógico, queda algo de complejidad, debido al gran tamaño de la base de datos. A muchos usuarios del sistema de base de datos no les preocupará toda esta información; en su lugar, tales usuarios necesitan acceder sólo a una parte de

la base de datos. Para que su interacción con el sistema se simplifique, se define la abstracción del nivel de vistas. El sistema puede proporcionar muchas vistas para la misma base de datos.

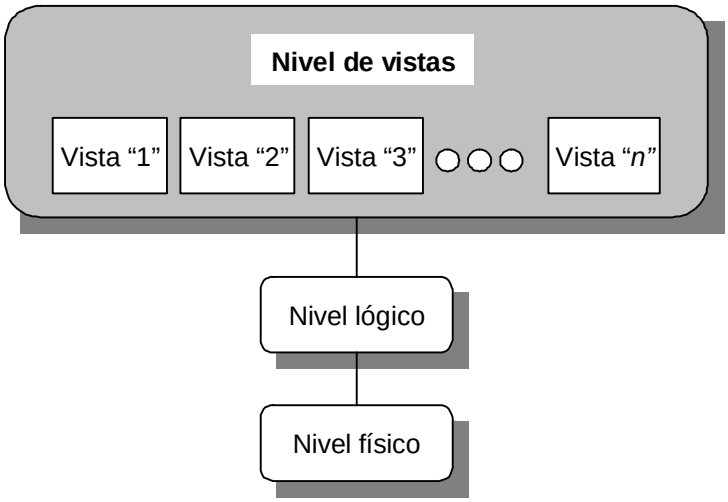

Figura 2.9. Los tres niveles de abstracción de datos.

Las bases de datos van cambiando con el tiempo conforme la información de actualiza. La colección de datos almacenada en un momento particular se llama un “Ejemplar” de la base de datos. El diseño completo de la base de datos se llama el “Esquema” de la base de datos. Los esquemas son raramente modificados, si es que lo son alguna vez. Los sistemas de bases de datos tienen varios esquemas divididos, de acuerdo con sus niveles de abstracción. En el nivel más bajo está el esquema físico, luego el esquema lógico, y en el nivel más alto se encuentra el esquema conceptual. En general, los sistemas de bases de datos soportan un esquema físico, un esquema lógico y varias vistas de usuarios.

La parte esencial de la estructura de base de datos es el modelo de datos: una colección de herramientas para describir los datos, las relaciones de datos, la semántica de los datos y las ligaduras de consistencia. Los modelos de datos se clasifican en:

- Modelos lógicos basados en objetos.
- Modelos lógicos basados en registros.

- Modelos físicos.

El modelo de datos basado en objetos se utiliza más para describir datos en los niveles lógicos y de vistas. Se caracteriza por el hecho de que proporcionan capacidades estructurales muy flexibles y permiten que las ligaduras de datos sean especificadas explícitamente. Dentro de los modelos lógicos basados en objetos se encuentra el Modelo Entidad – Interrelación (E-R) que está basado en una percepción del mundo real. Este modelo consta de una colección de objetos básicos llamados "entidades" y, de "interrelaciones" entre estos objetos. Una entidad es una «cosa» u «objeto» en el mundo real que es distinguible de otros objetos y de la cual se quiere almacenar información en la base de datos. Las entidades se describen mediante un conjunto de "atributos". Una interrelación es una asociación entre varias entidades. El conjunto de todas las entidades del mismo tipo y el conjunto de todas las relaciones del mismo tipo se denominan Conjunto de Entidades y Conjunto de Interrelaciones, respectivamente. Además de entidades e interrelaciones, el modelo E-R representa ciertas restricciones que los contenidos de la base de datos deben de cumplir. Una restricción muy importante es la correspondencia de la cardinalidad, que expresa el número de entidades con las que otra entidad se puede asociar a través de un conjunto de relaciones (Castaño y Piattini, 1999). La totalidad de estructuras lógicas de una base de datos se pueden expresar gráficamente mediante un diagrama E-R, que consta de los siguientes componentes:

- Rectángulos, que representan conjunto de entidades.
- Elipses, que representan atributos.
- Rombos, que representan las interrelaciones entre conjunto de entidades.
- Líneas, que unen los atributos con los conjuntos de entidades y los conjuntos de entidades con las interrelaciones. Estas líneas utilizarán algunas flechas de dirección para denotar el tipo de interrelación existente entre dos entidades.
- Cada componente se etiqueta con la entidad o interrelación que representa; vea Figura 2.10.

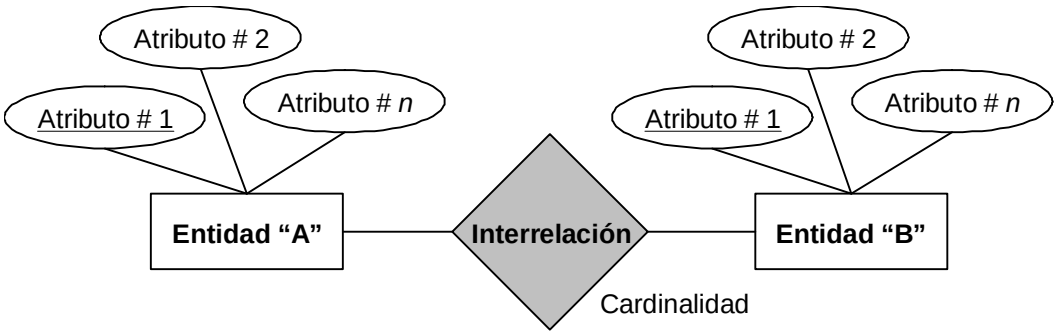

Figura 2.10. Un ejemplo de un diagrama Entidad Interrelación.

Los modelos lógicos basados en registros se utilizan para describir datos en los niveles lógicos y de vistas. En contraste con los modelos de datos basados en objetos, se usan tanto para especificar la estructura lógica completa de la base de datos como para proporcionar una descripción de alto nivel de la implementación. Los modelos basados en registros se llaman así debido a que la base de datos se estructura en registros de formato fijo de diferentes tipos de datos. En cada tipo de registro se define un número fijo de campos o atributos, y cada campo tiene normalmente una longitud fija. El uso de registros de longitud fija simplifica la implementación en el nivel físico de la base de datos. Dentro de este tipo de modelos el más utilizado es el Modelo Relacional. En este modelo, se usa una colección de tablas para representar tanto los datos como las relaciones en conjunto entre esos datos, y en donde a cada una se le asigna un nombre exclusivo. Cada tabla tiene varias columnas, y cada columna tiene un nombre único. Cada fila de una tabla representa una relación entre un conjunto de valores. En la Figura 2.11, se muestra un ejemplo. El modelo relacional relaciona registros mediante los valores que ellos contienen.

El modelo de datos físico se usa para describir datos en un nivel más bajo. Es el proceso para elegir estructuras de almacenamiento y caminos de acceso específicos para que los archivos de la base de datos tengan un buen rendimiento con las diversas aplicaciones de la base de datos (Elmasri y Navathe, 2000). Cada DBMS ofrece varias opciones de organización de archivos y caminos de acceso, entre ellas diversos tipos de indización, agrupamiento de registros relacionados en bloques de disco, enlace de registros relacionados mediante apuntadores y varias técnicas de dispersión.

Nombre-cliente	Número-cliente	Calle-cliente	Ciudad-cliente	Número-cuenta
González	19283746	Arenal	León	C-101
López	67789901	Mayor	Santiago	C-305
Santos	32112312	Carretas	Mazatlán	C-217
González	19283746	Arenal	León	C-222

Número-cuenta	Saldo
C-101	$ 100,000.00
C-305	$ 70,000.00
C-217	$ 150,000.00
C-222	$ 140,000.00

Figura 2.11. Un ejemplo de base de datos Relacional.

La capacidad de modificar una definición de esquema en un nivel sin que afecte a una definición de esquema en el siguiente nivel más alto se llama Independencia de Datos. Esta característica se logra en una base de datos relacional al utilizar una arquitectura de tres capas. Existen dos niveles de independencia de datos:

1. Independencia física de los datos. Es la capacidad para modificar el esquema físico sin provocar que los programas de aplicación tengan que rescribirse. Las modificaciones en el nivel físico ocasionalmente con necesarias para mejorar el rendimiento del sistema en general.
2. Independencia lógica de los datos. Es la capacidad para modificar el esquema lógico sin causar que los programas de aplicación tengan que rescribirse. Las modificaciones en el nivel lógico son necesarias siempre que la estructura lógica de la base de datos se altere.

La independencia lógica de datos es más difícil de proporcionar que la independencia de datos física, ya que los programas de aplicación son fuertemente dependientes de la estructura lógica de los datos a los que ellos acceden. La independencia de datos esconde los detalles de la implementación a los usuarios para permitirles

concentrarse en la estructura general, más que en los detalles de implementación de nivel más bajo.

En muchos DBMS en los que no se mantiene una separación estricta de niveles, el DBA y los diseñadores de la base de datos utilizan un mismo lenguaje, el Lenguaje de Definición de Datos o DDL (*Data Definition Language*), para definir ambos esquemas. El DBMS contará con un compilador de DDL cuya función será procesar enunciados escritos en el DDL para identificar las descripciones de los elementos de los esquemas y almacenar la descripción del esquema en el catálogo del DBMS. Cuando un DBMS tiene una clara separación entre los niveles conceptual e interno, el DDL servirá solamente para especificar el esquema conceptual.

Se utiliza otro lenguaje, el Lenguaje de Definición de Almacenamiento o SDL (*Storage Definition Language*) para especificar el esquema interno. Las correspondencias entre los dos esquemas se pueden especificar en cualquiera de los dos lenguajes. Para una verdadera arquitectura de tres esquemas se necesita un tercer lenguaje, el Lenguaje de Definición de Vistas o VDL (*View Definition Language*), para especificar las vistas del usuario y sus correspondencias con el esquema conceptual.

Una vez que se han compilado los esquemas de la base de datos y que en ésta se han introducido datos, los usuarios requerirán algún mecanismo para manipularla. Las operaciones de manipulación más comunes son la obtención, la inserción, la eliminación y la modificación de los datos. El DBMS ofrece un Lenguaje de Manipulación de Datos o DML (*Data Manipulation Language*) para estos fines. Son dos los principales tipos de DML. Los DML de alto nivel y el DML por procedimientos. El primero se puede utilizar de manera independiente para especificar operaciones complejas de base de datos en forma concisa. En muchos DBMS es posible introducir interactivamente instrucciones de DML de alto nivel desde una terminal o bien incorporados en un lenguaje de programación de propósito general. En el segundo caso, es preciso identificar los enunciados de DML dentro del programa para que el DBMS pueda procesarlos. Los DML de bajo nivel o por procedimientos deben estar incorporados en un lenguaje de programación de propósito general. Por lo regular, este tipo de DML obtiene registros individuales de la base de datos y los procesa por separado; por tanto, necesita utilizar elementos del lenguaje de programación, como la creación de ciclos, para obtener y procesar cada registro individual

de un conjunto de registros. Por esta razón, los DML de bajo nivel se conocen como DML de registro por registro. Siempre que las órdenes de un DML, sean de alto o de bajo nivel, se incorporen en un lenguaje de programación de propósito general, a ese lenguaje se le llamará lenguaje anfitrión, y al DML, sublenguaje de datos.

Dentro de una base de datos, la Transacción es lo más importante (Silberschatz, et. al. 1998). Una Transacción es una colección de operaciones que se lleva a cabo como una función lógica simple en una aplicación de bases de datos. Cada Transacción es una unidad de atomicidad, consistencia y durabilidad. Así, se requiere que las transacciones no violen ninguna ligadura de consistencia de la base de datos. Es decir, si la base de datos era consistente cuando la transacción comenzó, la base de datos debe ser consistente cando la transacción termine con éxito. Asegurar estas propiedades de atomicidad y durabilidad es responsabilidad del componente del DBMS que se encargue de la Gestión de Transacciones. En un sistema distribuido se dan dos tipos de transacciones, las locales y las globales. Una transacción local es aquella que accede a los datos del único emplazamiento en el cual se inició la transacción, por ejemplo, el Emplazamiento “A” busca información dentro de sí mismo; una transacción global es aquella que o bien accede a los datos situados en un emplazamiento diferente de aquel en el que se inició la transacción, o bien accede a datos de varios emplazamientos distintos, por ejemplo, el Emplazamiento “A” busca información dentro del Emplazamiento “B” y/o del Emplazamiento “C”, etcétera.

Teniendo estos componentes, herramientas y recursos podemos decir entonces que el DBMS es el *software* que maneja todo acceso a la base de datos. De manera conceptual, lo que sucede al momento de realizar algún proceso en ella es lo siguiente:

1. Un usuario emite una petición de acceso, utilizando algún sublenguaje de datos específico (por lo regular, SQL).
2. El DBMS intercepta esa petición y la analiza.
3. El DBMS inspecciona el esquema externo para ese usuario, la transformación externa/conceptual correspondiente, el esquema conceptual, la transformación conceptual/interna y la definición de la estructura de almacenamiento.
4. El DBMS ejecuta las operaciones necesarias sobre la base de datos almacenada.

En realidad, el sistema de bases de datos se divide en ciertos módulos que se encargan de cada una de las responsabilidades del sistema completo. Algunas de estas funciones del sistema de bases de datos las proporciona el sistema operativo de la computadora. En la mayoría de los casos los OS de la computadora proporcionan sólo los servicios más básicos y los sistemas de bases de datos deben construirse sobre esta base. Así, el diseño de un sistema de bases de datos debe incluir consideraciones de la interfaz entre el OS y el mismo. El DBMS interactúa con el OS cuando requiere acceso al disco (a la base de datos o al catálogo). Si muchos usuarios comparten el mismo sistema de cómputo, el OS programará las solicitudes de acceso a disco del DBMS junto con otros procesos. El DBMS también se comunica con los compiladores de los lenguajes de programación anfitriones, y puede ofrecer interfaces con el usuario como ayuda para cualquiera de los tipos de usuarios que intenten acceder al DBMS.

Además se necesitan varias estructuras de datos como parte de la implementación física del sistema:

- Archivos de datos, que almacenan la base de datos en sí en el disco duro.
- Catálogo del sistema o Diccionario de datos, que almacena meta datos acerca de la estructura de la base de datos en el disco duro. El diccionario de datos se usa mucho. Por lo tanto, se debería poner gran énfasis en el desarrollo de un buen diseño e implementación eficiente del diccionario. Almacena la información sobre los nombres de los archivos y de los elementos de información, los detalles de almacenamiento de cada archivo, la información de correspondencia entre los esquemas y las restricciones. Los módulos del DBMS que necesiten conocer esta información deberán consultarlo.
- Índices, que proporcionan acceso rápido a elementos de datos que tienen valores particulares.

El DBMS debe interactuar (en ciertos casos) con un *software* de comunicaciones, cuya función es permitir que las peticiones emitidas a la base de datos por parte de los usuarios situados en lugares remotos respecto al sistema de base de datos, tengan acceso a

éste a través de computadoras en red; peticiones en forma de “Mensajes de comunicación”. De forma similar, las respuestas que regresan del DBMS hacia la estación de trabajo del usuario, también son transmitidas en forma de dichos mensajes. Todas estas transmisiones de mensajes se llevan a cabo bajo el control de otro componente de *software*, el Administrador de Comunicaciones de Datos o DCM (*Data Communications Manager*). Este administrador no forma parte de un DBMS como tal, sino que es un sistema autónomo; sin embargo, es necesario que trabaje en armonía con él. Este funcionamiento conjunto recibe el nombre de Sistema de DB/DC (*Data Base/Data Communications*).

Por añadidura, algunos sistemas distribuidos están físicamente dispersos en varias computadoras. En este caso, se requieren redes de comunicaciones para conectar las computadoras. Según el número de sitios en los que está distribuida la base de datos, se pueden dividir a los DBMS como centralizados; esto es, sus datos se almacenan en una sola computadora. Los DBMS centralizados pueden atender a varios usuarios, pero el DBMS y la base de datos en sí, residen por completo en una sola computadora. En los DBMS de este tipo, el DBMS y la base de datos están colocadas en varios sitios, conectados por una red de computadoras. Los DBMS homogéneos utilizan el mismo *software* de DBMS en múltiples sitios. Por ejemplo, en un sistema de bases de datos distribuido homogéneo tiene que tener las siguientes características:

- Los distintos emplazamientos están informados sobre el estado actual de funcionamiento de los demás.
- Aunque algunas relaciones (por ejemplo, de una base de datos) pueden estar almacenadas sólo en algunos emplazamientos, éstos comparten un esquema global común.
- Cada emplazamiento proporciona un entorno para la ejecución de transacciones locales y globales.
- En cada emplazamiento se ejecuta el mismo *software* de gestión de bases de datos distribuidas.

Una tendencia reciente consiste en crear *software* para tener acceso a varias bases de datos autónomas preexistentes almacenadas en DBMS heterogéneos. Esto da lugar a los

DBMS federados en los que los DBMS participantes están acoplados con un cierto grado de autonomía local y en dónde manejar las transacciones globales son más laboriosas. Muchos DBMS utilizan la arquitectura Cliente/Servidor. Este término se usa para caracterizar a un DBMS cuando la aplicación se ejecuta físicamente en una computadora llamada Cliente, y cuando otra computadora llamada Servidor se encarga del almacenamiento y acceso a los datos (Pressman, 2002).

II.2.1. Bases de datos distribuidas.

Desde el punto de vista de los sistemas distribuidos, un sistema de base de datos puede ser apreciado como un sistema que tiene una estructura muy sencilla de dos partes, las cuales consisten en un Servidor y un Cliente.

- El Servidor es precisamente el propio DBMS. Soporta todas las funciones básicas del DBMS como la definición de datos, manipulación de datos, seguridad e integridad de los datos, etcétera. Proporciona todo el soporte de los niveles externo, conceptual e interno.
- Los Clientes son las diversas aplicaciones que se ejecutan sobre el DBMS, tanto aplicaciones escritas por el usuario como aplicaciones integradas (es decir, aplicaciones proporcionadas por el fabricante del DBMS o por alguna otra compañía). Por supuesto, en lo que concierne al Servidor, no hay diferencia entre las aplicaciones escritas por el usuario y las integradas; todos usan la misma interfaz con el Servidor.

Puesto que el sistema en su conjunto puede ser dividido en dos partes (Clientes y Servidores), surge la posibilidad de operar en modo local (o sea que los componentes residan en la misma computadora) o en modo distribuido; vea Figura 2.12.

Procesamiento distribuido significa que es posible conectar distintas computadoras en una red de comunicaciones de tal manera que una tarea de procesamiento de datos pueda dividirse entre varias computadoras de la red. De hecho, esta posibilidad es atractiva por razones económicas (principalmente) que el término “Cliente/Servidor” ha

llegado a aplicarse exclusivamente al caso en el que el Cliente y el Servidor están en computadoras distintas. También existe la posibilidad de que en lugar de una sola computadora Cliente se conecte con su Servidor pueda darse el caso de que ahora a una computadora Servidor permita la conexión con varias computadoras Cliente dentro de una red de comunicaciones. Sin embargo, también es muy común actualmente de que las computadoras Cliente, no nada más acceden a su propio Servidor sino que, también se conectan a otros servidores dentro de la misma red, es decir, los clientes podrían tener almacenados sus propios datos y que la máquina Servidor podría tener sus propias aplicaciones. Por lo tanto es común que cada máquina actúe como Servidor para ciertos usuarios y como Cliente para otros. Cada computadora soportaría entonces un sistema de bases de datos completo; vea la Figura 2.13. La idea final es que una sola computadora Cliente podría ser capaz de acceder a varias máquinas servidores diferentes.

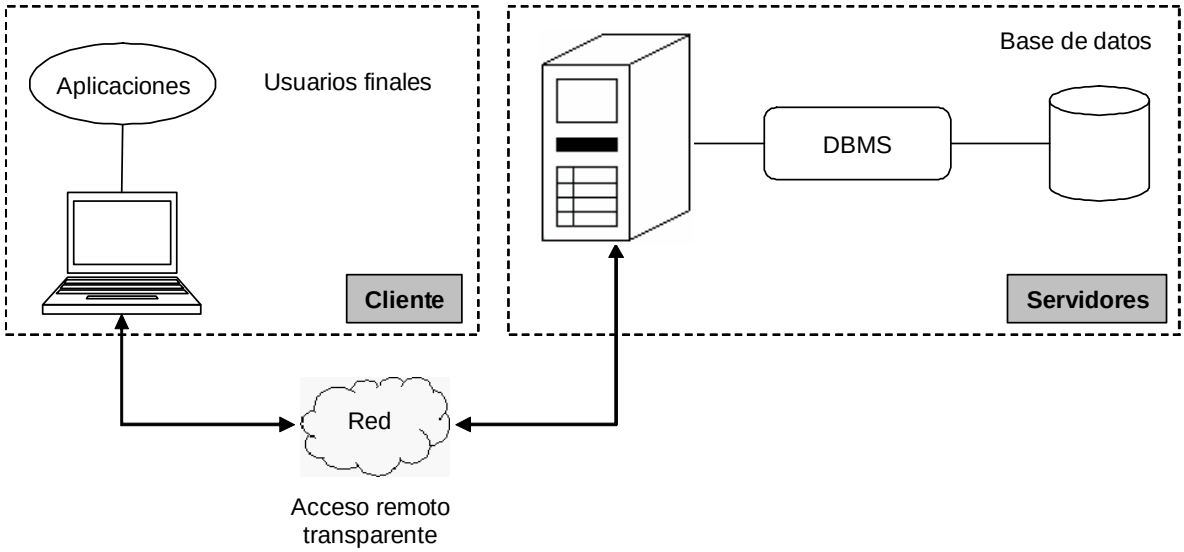

Figura 2.12. Cliente y Servidor de un Sistema de Bases de Datos.

Esta posibilidad es conveniente ya que la totalidad de los datos no están almacenados en una sola computadora, sino más bien están esparcidos a través de muchas computadoras distintas, y las aplicaciones necesitarán a veces tener la posibilidad de acceder a los datos de más de una computadora. Esto se puede proporcionar en dos formas:

- Una computadora Cliente dada podría ser capaz de acceder a cualquier cantidad de servidores, pero sólo uno a la vez (es decir, cada petición individual de base de datos debe ser dirigida a un solo Servidor). En un sistema así, no es posible combinar datos de dos o más servidores con una sola petición. Además, el usuario de dicho sistema debe saber qué computadora en particular contiene qué piezas de datos.
- El Cliente podría ser capaz de acceder a varios servidores en forma simultánea (es decir, una sola petición de base de datos podría combinar datos de varios servidores). En este caso, los servidores ven al Cliente – desde un punto de vista lógico – como si en realidad fuera un solo Servidor y el usuario no tiene que saber qué computadoras contienen qué piezas de datos. (Date, 2001).

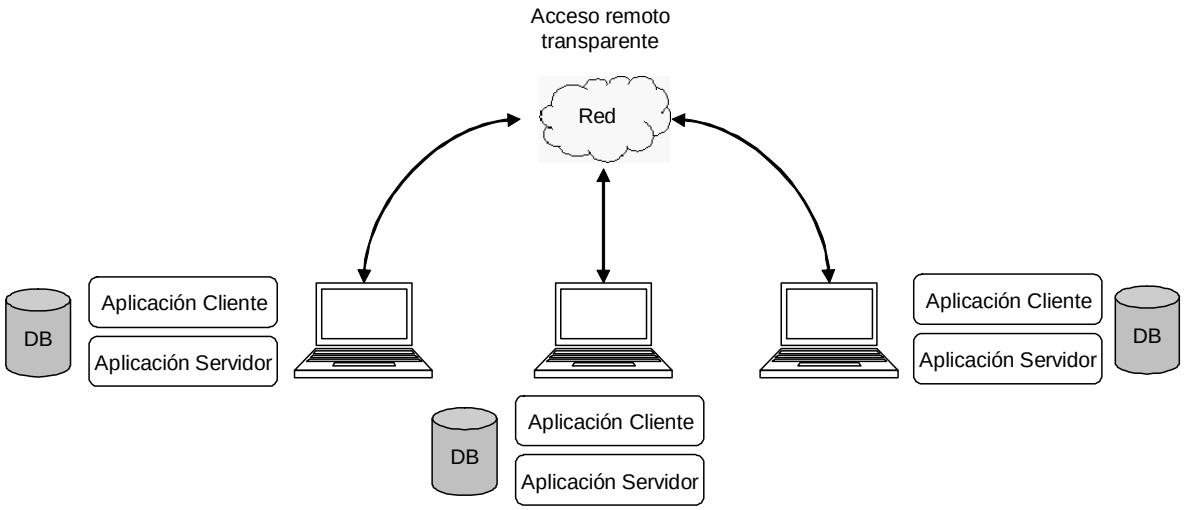

Figura 2.13. Cada computadora opera como Cliente y como Servidor.

Este último caso constituye a un **Sistema de Bases de Datos Distribuida**. La base de datos distribuida debe ser capaz de operar de "manera transparente" sobre los datos que están dispersos a través de una variedad de DBMS distintos que operan en computadoras diferentes que a su vez son manejadas por sistemas operativos diversos y, que están interconectadas por una variedad de redes de comunicación. De manera transparente significa que la aplicación opera desde un punto de vista lógico como si los datos fueran manejados por un solo DBMS y en una sola computadora.

La base de datos distribuida en realidad es un tipo de base de datos “virtual” cuyas partes componentes están almacenadas en varias bases de datos “reales” distintas que se encuentran en varios sitios distintos (de hecho, es la unión lógica de esas bases de datos reales). Cada sitio tiene sus propias bases de datos reales, sus propios usuarios locales, su propio DBMS local y *software* de administración de transacciones (incluyendo su propio *software* local para bloqueo, registro de bitácora, recuperación, etcétera), así como su propio administrador de DB/DC local. En particular, un usuario determinado puede realizar operaciones sobre los datos desde su propio sitio local, tal como si ese sitio no participara nunca en el sistema distribuido.

La ventaja principal de los sistemas distribuidos de bases de datos es que permite reflejar la estructura de una organización – los datos locales sin conservados localmente en el lugar en donde pertenecen de manera más lógica – y al mismo tiempo, permite tener acceso a datos remotos cuando sea necesario. Para que un sistema distribuido sea exitoso, debe ser relacional. La tecnología relacional es un requisito previo para la tecnología distribuida efectiva cuando se utilicen las bases de datos (Date, 2001).

El sistema de base de datos distribuida puede ser considerado como un tipo de sociedad entre los DBMS locales en cada uno de los sitios locales. Una extensión del DBMS local, proporciona la funcionalidad necesaria y es la combinación de este componente con el DBMS, lo que constituye el Sistema de Administración para Bases de Datos Distribuida o D-DBMS (*Distribuited Data Base Management System*).

El término D-DBMS puede describir diversos sistemas que presentan muchas diferencias entre sí. El punto principal de todos estos sistemas tiene en común el hecho de que los datos y el *software* están distribuidos entre múltiples sitios conectados por alguna especie de red de comunicaciones.

El primer factor que se considera para la clasificación de los sistemas de bases de datos distribuidos es el grado de **homogeneidad** del *software* del D-DBMS. Si todos los servidores (o DBMS locales e individuales) utilizan *software* idéntico y todos los clientes emplean *software* idéntico, se dice que el D-DBMS es homogéneo; en caso contrario se le caracteriza como **heterogéneo**. Otro factor relacionado con el grado de homogeneidad es el grado de autonomía local. Si todo acceso al D-DBMS debe hacerse a través de un Cliente,

el sistema no tiene autonomía local. Por otro lado, si se permite a las transacciones locales acceso directo a un Servidor, el sistema tendrá cierto grado de autonomía local.

En un extremo de la gama de autonomía tenemos un D-DBMS que da al usuario la impresión de ser un sistema de base de datos centralizado. Solo hay un esquema conceptual, y todo acceso al sistema se hace a través de un Cliente, de modo que no hay autonomía local. En el otro extremo nos encontramos con un tipo de D-DBMS denominado Sistema de Bases de Datos Federado. En un sistema así, cada Servidor es un DBMS centralizado independiente y autónomo que tiene sus propios usuarios locales, transacciones locales y DBA, y por ende, posee un alto grado de autonomía local. Cada Servidor puede autorizar el acceso a porciones específicas de su base de datos definiendo un esquema de exportación, el cual especifica la parte de la base de datos a la cual puede tener acceso cierta clase de usuarios no locales. En esencia, un Cliente en un sistema así es una interfaz adicional de usuario para varios servidores (DBMS locales) que permite a un usuario de multibases de datos (o global) tener acceso a información almacenada en varias de estas bases de datos autónomas. Observe entonces que un sistema federado es un híbrido entre los sistemas centralizados con los distribuidos; es un sistema centralizado para los usuarios autónomos locales y es un sistema distribuido para los usuarios globales.

Un tercer aspecto que puede servir para clasificar las bases de datos distribuidas es el grado de transparencia de la distribución o, de manera alternativa, el grado de integración de los esquemas. Si el usuario percibe un solo esquema integrado sin información alguna relativa a la fragmentación, replicación o distribución, se dice que el D-DBMS tiene un alto grado de transparencia de distribución (o de integración de esquemas). Por otro lado, si el usuario puede ver toda la fragmentación, el reparto y la replicación, el D-DBMS no tiene transparencia de distribución ni integración de esquemas. En este caso, el usuario debe hacer referencia a copias específicas de fragmentos en sitios específicos cuando formule una consulta, anexando el nombre del sitio como prefijo de cada nombre de relación o fragmento.

Esto forma parte del problema de dar nombres en los sistemas distribuidos. Hay casos en los que el D-DBMS no ofrece transparencia de distribución; si es así, es responsabilidad del usuario especificar sin ambigüedad el nombre de una copia de relación o de fragmento en particular. La tarea es más difícil en un sistema de multibases de datos,

porque es de suponer que cada Servidor (DBMS local) se creó de manera independiente, y en consecuencia es muy posible que se hayan usado nombres conflictivos en diferentes servidores. En el caso de un D-DBMS que provee un esquema integrado, en cambio, la asignación de nombres se convierte en un problema interno del sistema, porque el usuario percibe un solo esquema sin las ambigüedades. El D-DBMS debe almacenar en el catálogo de distribución todas las correspondencias entre los objetos del esquema integrado y los objetos distribuidos en los diversos DBMS que conforman el sistema distribuido.

En general, el principio fundamental de una base de datos distribuida cualquiera ofrece doce consideraciones que se deben de tomar en cuenta en una base de datos de este tipo:

- Autonomía local.
- No-dependencia de un sitio central.
- Operación continúa.
- Independencia de ubicación.
- Independencia de fragmentación.
- Independencia de replicación.
- Procesamiento de consultas distribuidas.
- Administración de transacciones distribuidas.
- Independencia del *hardware*.
- Independencia del OS.
- Independencia de red.
- Independencia del DBMS.

La AUTONOMÍA LOCAL se refiere que los sitios en un sistema distribuido deben de ser autónomos. Significa que todas las operaciones en un sitio dado están controladas por ese sitio; ningún sitio “X” debe depender de algún sitio “Y” para su operación satisfactoria (ya que de lo contrario, cualquier falla en el sitio “Y” podría significar que el sitio “X” no pueda ejecutar operaciones, aun cuando no haya nada malo en el propio sitio “X”, lo que es una situación indeseable). La autonomía local también implica que los datos locales son poseídos y administrados localmente con contabilidad local: todos los datos pertenecen

“realmente” a alguna base de datos local, aun cuando estén accesibles desde otros sitios remotos. Por lo tanto, la seguridad, integridad y representación de almacenamiento de los datos locales permanecen bajo el control y la jurisdicción del sitio local. Sin embargo, la autonomía total no es totalmente alcanzable; existen varias situaciones en las que un sitio “X” debe de transferir un determinado de control a algún otro sitio “Y”. Por lo tanto, el objetivo de la autonomía queda establecido con mayor precisión como: los sitios deben de permanecer autónomos en el mayor grado posible.

La autonomía local implica que todos los sitios deben ser tratados como iguales. Por lo tanto, no debe haber particularmente dependencia de un sitio “maestro” central para algún servicio central tal que todo ese sistema dependa de ese sitio central. Por lo tanto, esto se logra si la autonomía local funciona. Pero la NO-DEPENDENCIA DE UN SITIO CENTRAL es necesaria por sí misma, aunque no se logre la autonomía local completa. La dependencia de un sitio central es indeseable por las siguientes dos razones: si el sitio central presenta un cuello de botella o un sobrecargo de trabajo y, el sistema sería vulnerable; es decir, si el sitio central falla, también fallará todo el sistema.

Por otra parte, una ventaja de los sistemas distribuidos es que deben de proporcionar mayor confiabilidad y mayor disponibilidad para obtener una OPERACIÓN CONTINUA:

- La confiabilidad, es decir, la probabilidad de que el sistema esté listo y funcionando en cualquier momento es aceptable ya que los sistemas distribuidos pueden continuar operando en un nivel reducido cuando hay alguna falla en algún componente independiente, tal como un sitio individual.
- La disponibilidad, es decir, la probabilidad de que el sistema esté listo y funcionando continuamente a lo largo de un periodo especificado también es buena, debido a la posibilidad de replicación de datos.

Lo anterior aplica al caso en el que ocurre un apagado no planeado (falla de algún tipo) en algún punto del sistema. Los apagados de este tipo son indeseables pero es muy difícil evitarlos. Por el contrario, los apagados planeados nunca deberían ser requeridos, es decir, nunca debería ser necesario apagar el sistema para realizar alguna tarea como la

incorporación de un nuevo sitio o la actualización del DBMS a una nueva versión en un sitio existente.

La idea básica de la INDEPENDENCIA DE UBICACIÓN (transparencia) es simple. Los usuarios no tienen que saber dónde están almacenados físicamente los datos, sino que deben ser capaces de comportarse – al menos desde un punto de vista lógico – como si todos los datos estuvieran almacenados en su propio sitio local. La independencia de ubicación es necesaria debido a que simplifica los programas de usuario y las actividades terminales. Permite que los datos emigren de un sitio a otro sin invalidar ninguno de estos programas o actividades. Dicha capacidad de emigración es necesaria, ya que permite mover los datos por la red en respuesta a los diferentes requerimientos de desempeño.

Un sistema soporta la fragmentación de datos cuando una relación dada puede ser dividida en partes o fragmentos, para efectos de almacenamiento físico. La fragmentación es necesaria por razones de rendimiento: los datos pueden estar almacenados en la ubicación donde son usados más frecuentemente para que la mayoría de las operaciones sean locales y se reduzca el tráfico en la red.

Existen básicamente dos tipos de fragmentación, horizontal y vertical, que corresponden a las operaciones relacionales de restricción y proyección, respectivamente. En términos generales, un fragmento puede ser derivado mediante una combinación de restricciones y proyecciones. Los fragmentos que se obtengan de una relación, deben ser independientes entre sí, en el sentido de que ninguno de ellos puede ser derivado a partir de los otros ni tienen una restricción o proyección que puede ser derivada a partir de los otros. La reconstrucción de la relación original a partir de los fragmentos es lograda por medio de las operaciones de Reunión y de Unión adecuadas, pertenecientes al álgebra relacional (Elmasri y Navathe, 2000). La facilidad de fragmentación y la de reconstrucción son dos razones principales por lo que los sistemas distribuidos de base de datos son relacionales; el modelo relacional proporciona exactamente las operaciones necesarias para esas tareas.

El Álgebra Relacional fue definida por primera vez por Codd en su artículo de 1970 y consta de un conjunto de ocho operadores que generan una especie de vistas de los datos que se encuentran almacenados dentro de ciertas relaciones y que cumplen con una condición específica. Estos operadores son:

- Selección (σ): Regresa una relación que contiene todas las tuplas de una relación en especial que satisface una condición especificada.
- Proyección (π): Regresa una relación que contiene todas las tuplas de una relación en especial después de quitar los atributos no especificados en la operación, es decir, únicamente nos devuelve los datos que nos interesan de una relación.
- Unión ($\cup$): El resultado de esta operación, denotado por $R \cup S$ es una relación que incluye todas las tuplas que están en la relación *R* o en *S* o en ambas. Las tuplas repetidas se eliminan.
- Intersección ($\cap$): El resultado de esta operación, denotado por $R \cap S$ es una relación que incluye las tuplas que están tanto en *R* como en *S*.
- Diferencia (-): El resultado de esta operación, denotado por *R* - *S* es una relación que incluye todas las tuplas que están en *R* pero no en *S*.
- Producto cartesiano ($\times$): Sirve para combinar tuplas de dos relaciones para poder identificar las tuplas relacionadas entre sí, es decir, crea una nueva relación con los todos los atributos de las dos relaciones combinando todas las tuplas entre sí.
- Reunión ($\bowtie$): Sirve para combinar tuplas relacionadas de dos relaciones en una sola tupla siempre que la condición de reunión sea satisfecha. Esta operación es muy importante en cualquier base de datos relacional que comprenda más de una relación, porque permite procesar los vínculos entre las relaciones. Cuando una operación de reunión combina a dos relaciones que tienen el mismo nombre en el atributo que se encuentra en la condición, se le conoce como Reunión natural ($*$).
- División ($\div$): La división de una relación *R* (llamada dividendo) por otra relación *S* (llamada divisor) es una relación *T* (llamada cociente) tal que, al realizarse su combinación con el divisor, todas las tuplas se encuentran en el dividendo.

Un sistema que soporta la fragmentación de datos también debe de soportar la INDEPENDENCIA DE FRAGMENTACIÓN (transparencia de fragmentación); es decir, los usuarios deben ser capaces de comportarse, al menos desde un punto de vista lógico, como si los datos en realidad estuvieran sin fragmentación alguna. Esta característica es necesaria debido a que simplifica los programas de usuario y las actividades terminales. Permite que los datos sean fragmentados en cualquier momento (y que sean redistribuidos en cualquier

momento) en respuesta a los diferentes requerimientos de rendimiento, sin invalidar ningún programa de usuario. La independencia de fragmentación implica que a los usuarios se les presentará una vista de los datos en la cual los fragmentos estarán recombinados lógicamente por medio de juntas y de uniones adecuadas. Es responsabilidad del "Optimizador del Sistema" determinar cuáles fragmentos necesitan ser accedidos físicamente para satisfacer cualquier solicitud de usuario dada, por medio del catálogo o el diccionario de datos.

Un sistema soporta la REPLICACIÓN DE DATOS cuando una relación almacenada dada – o un fragmento dado de una relación guardada – puede ser representada por muchas copias distintas, o réplicas, guardadas en muchos sitios distintos. Las réplicas son necesarias por dos razones. Primero, puede significar un mejor rendimiento (las aplicaciones pueden operar sobre copias locales en lugar de tener que comunicarse con otros sitios remotos). Segundo, también puede significar una mejor disponibilidad (un objeto replicado dado que permanece disponible para su procesamiento – al menos para su recuperación – mientras esté disponible al menos una copia). Por supuesto, la principal desventaja de la replicación es que al actualizar un objeto replicado dado que es necesario actualizar todas las copias de ese objeto: el problema de la propagación de la actualización. La replicación debe ser "transparente para el usuario". Un sistema que soporte la replicación de datos también debe de soportar la independencia de la replicación; es decir, los usuarios deben ser capaces de comportarse – al menos desde un punto de vista lógico – como si los datos en realidad no estuvieran replicados. La independencia de la replicación (al igual que la independencia de la ubicación y la de fragmentación) es necesaria debido a que simplifica los programas de usuario y las actividades terminales; en particular, permite que las réplicas sean creadas y destruidas en cualquier momento en respuesta a los distintos requerimientos, sin invalidar ninguno de esos programas de usuario o actividades. La independencia de replicación implica que es responsabilidad del optimizador del sistema determinar cuáles réplicas necesitan ser accedidas físicamente para satisfacer cualquier solicitud de usuario dada.

El PROCESAMIENTO DE CONSULTAS hace referencia a la serie de actividades implicadas en la extracción de datos en una base de datos. Estas actividades incluyen la traducción de consultas expresadas en lenguajes de bases de datos de alto nivel en

expresiones implementadas en el nivel físico del sistema, así como transformaciones de optimización de consultas y la evaluación real de las mismas. El costo del procesamiento de una consulta está determinado por la cantidad de veces que se accede al disco duro de la computadora, comparativamente más lento que el acceso a la memoria principal. Normalmente hay muchas estrategias posibles para procesar una consulta dada, especialmente si la consulta es compleja. La diferencia entre una buena y una mala, en términos del número de accesos a disco que necesitan, es a menudo importante y podría ser de varios órdenes de magnitud[6]. Por esta razón, vale la pena que el sistema gaste una cantidad sustancial de tiempo en la selección de una buena estrategia para el procesamiento de la consulta, aunque ésta se ejecute solamente una vez (Date, 2001). Los pasos involucrados para el procesamiento de una consulta (Figura 2.14) son:

- Análisis y traducción.
- Optimización.
- Evaluación.

Antes de empezar el procesamiento de una consulta, el sistema debe traducir la consulta a una forma utilizable. Un lenguaje como SQL es adecuado para el uso humano, pero es poco apropiado para una representación interna en el sistema de la consulta, así una representación interna más útil está basada en el álgebra relacional. Así, la primera acción que el sistema tiene que hacer para procesar una consulta es la traducción de la consulta dada a su formato interno. Este proceso de traducción es similar al trabajo que realiza el analizador de un compilador. Durante la generación del formato interno de una consulta el analizador comprueba la sintaxis de la consulta del usuario, verifica que los nombres de las relaciones que aparecen en la consulta sean nombres de relaciones de la base de datos. Luego se construye un árbol para el análisis de la consulta, que se transformará en una expresión del álgebra relacional. Si la consulta estuviera expresada en términos de una

[6] Se refiere a los costos de cómputo necesarios para resolver un problema, en este caso la cantidad de accesos y de tiempo en los procesos de lectura/escritura hacia la unidad de disco duro de una computadora. (Silberschatz, 1998).

vista, la fase de traducción también sustituye todos los usos de la vista mediante expresiones del álgebra relacional que definen la vista.

En los sistemas distribuidos hay que tener en cuenta otros aspectos más que el acceso al disco:

- El costo de la transmisión de los datos por la red.
- La ganancia potencial en rendimiento respecto que hacer que varios emplazamientos procesen en paralelo parte de la consulta.

Estos factores varían ampliamente con el tipo de red sobre el que esté construido el sistema distribuido y la velocidad de los discos duros de las computadoras. Lo recomendable es encontrar el equilibrio adecuado entre ambos.

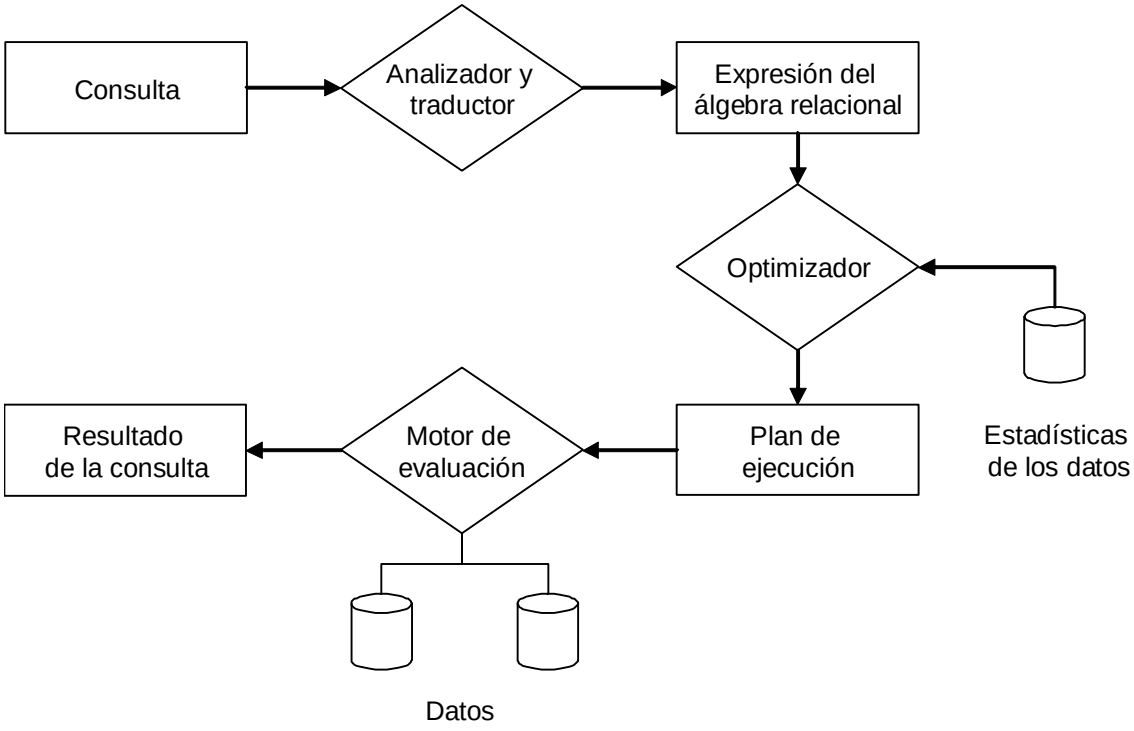

Figura 2.14. Pasos en el procesamiento de una consulta.

A menudo, desde el punto de vista del usuario de una base de datos, se considera a un conjunto de varias operaciones sobre una base de datos como una única operación. Se

llama "Transacción" a una colección de operaciones que forman una única unidad lógica de trabajo. Un sistema de base de datos debe asegurar que la ejecución de las transacciones se realice adecuadamente, a pesar de la existencia de fallos, o se ejecuta la transacción completa o no se ejecuta en absoluto. Además debe de administrar la ejecución concurrente de las transacciones evitando introducir inconsistencias en los datos del sistema. La ADMINISTRACIÓN DE TRANSACCIONES DISTRIBUIDAS consiste en dos elementos: el control de la recuperación y el control de la concurrencia. En los sistemas distribuidos, una sola transacción puede involucrar la ejecución de código en muchos sitios; en particular puede involucrar actualizaciones en muchos emplazamientos. Por lo tanto, decimos que cada transacción consiste en varios “agentes”, donde un agente es el proceso realizado en nombre de una transacción dada en un sitio dado. Y el sistema necesita saber cuando dos agentes son parte de la misma transacción; por ejemplo, no se debe permitir que dos agentes que son parte de la misma transacción caigan en un “bloqueo mortal” entre ellos. En el control de recuperación, para asegurarse de que una transacción dada sea atómica (todo o nada) en el ambiente distribuido, el sistema debe por lo tanto asegurarse de que el conjunto de agentes para esa transacción sea confirmado o deshecho al unísono. Este efecto puede lograrse por medio del protocolo de confirmación de dos fases. En la mayoría de los sistemas distribuidos el control de concurrencia está basado generalmente en el bloqueo de ciertos elementos de la base de datos.

La INDEPENDENCIA DEL *HARDWARE* quiere decir que en una red de comunicaciones puede ser que existan computadoras que contengan diferentes tipos de arquitecturas de red o que tengan diferentes tipos de microprocesadores que son incompatibles entre sí como, por ejemplo: una red de tres computadoras y cada una con microprocesadores diferentes. Este objetivo de las bases de datos distribuidas implica que existe una necesidad real de poder integrar los datos en todos esos sistemas y presentar al usuario una “imagen de un sistema único”. Por lo tanto, es necesario tener la posibilidad de ejecutar el mismo DBMS (homogeneidad) o diferentes (heterogeneidad) en diversas plataformas de *hardware*, y además, hacer que esas computadoras participen como socios igualitarios en un sistema distribuido.

La INDEPENDENCIA DE SISTEMA OPERATIVO es parte del objetivo anterior. Obviamente es necesario no sólo tener la posibilidad de ejecutar un mismo o diferente

DBMS en diferentes plataformas de *hardware*, sino también ejecutarlos en diferentes plataformas de sistema operativo.

En el caso de la INDEPENDENCIA DE RED; si el sistema va a tener la posibilidad de soportar muchos sitios distintos – con *hardware* distinto y OS diferentes – es obviamente necesario tener la posibilidad de soportar también una variedad de redes de comunicación distintas, es decir, no necesariamente tienen que estar conectados estos emplazamientos con cables para redes sino que también pueden estar unidas, por ejemplo: vía infrarroja.

INDEPENDENCIA DEL DBMS. El sistema distribuido tiene que estar construido y ser funcional, ya sea con DBMS homogéneos o heterogéneos. Al final de cuentas, el nivel de abstracción lógico/conceptual tiene que ser el mismo, la replicación de datos y los demás procesos de un DBMS distribuido tienen que funcionar de manera similar, al menos en cierto grado. El soporte de la heterogeneidad es necesario. El hecho es que por lo general las instalaciones de computación en realidad emplean no sólo muchas computadoras diferentes y muchos sistemas operativos de red de igual manera, sino que muy frecuentemente, también DBMS de diversos fabricantes; y sería muy bueno si esos sistemas de bases de datos pudieran participar de alguna forma en un sistema distribuido. En otras palabras, el sistema distribuido ideal debe proporcionar una independencia del DBMS.

II.2.2. Servidores de bases de datos.

Dentro de los diferentes tipos de servidores más comunes que existen dentro de la tecnología Cliente/Servidor, están los Servidores de Bases de Datos. Los servidores de este tipo son el modelo dominante para la creación de aplicaciones Cliente/Servidor y también dentro de las aplicaciones distribuidas. Quizá la tendencia más importante entre los servidores de bases de datos de cualquier tamaño sea la aparición del SQL como la forma estándar de manipular, definir y controlar los datos dentro de una base de datos. SQL es un poderoso lenguaje declarativo y orientado a conjuntos que consiste en unos cuantos comandos potentes y flexibles, para la manipulación de la información recogida en tablas y fue creado para sistemas de bases de datos que siguen el modelo relacional.

El modelo relacional de datos fue desarrollado por Edgar Frank Codd en IBM a principios de la década de los setenta con un enorme sustento matemático de la teoría de

conjuntos y el cálculo de predicados incluyendo el álgebra relacional[7]. Mediante SQL, se manipulan y se controlan al mismo tiempo un conjunto de registros. Le indica al Servidor de bases de datos de SQL, qué datos necesitan, tras de lo cual éste se las ingenia para conseguirlos. El modelo relacional supone una clara distinción entre los aspectos físicos de los datos y su representación lógica. Las cosas se realizan de una manera que los datos parezcan simples tablas, tras las cuales se oculta la complejidad de los mecanismos de acceso al almacenamiento. Este modelo le evita tener que ocuparse de los detalles del modo de almacenamiento de los datos y vuelve estrictamente lógico el procedimiento de acceso a ellos. Mediante el empleo de instrucciones SQL todo lo que usted tiene que hacer es especificar las tablas, columnas y calificadores de filas para obtener los datos que necesita.

El lenguaje SQL sirve para realizar complejas operaciones de datos con unos cuantos comandos simples en situaciones para las que se habría requerido cientos de líneas de código (de un lenguaje de programación de propósito general) para realizarlas. Por ser un lenguaje declarativo de alto nivel o de no-procedimiento, que gracias a su fuerte base teórica y su orientación al manejo de conjuntos de registros, y no a registros individuales, permite una alta productividad en codificación.

El SQL permite fundamentalmente dos modos de uso:

- Un uso interactivo, destinado principalmente a los usuarios finales avanzados u ocasionales, en el que las diversas sentencias SQL se escriben y ejecutan en línea de comandos, o un entorno semejante.
- Un uso integrado, destinado al uso por parte de los programadores dentro de programas escritos en cualquier lenguaje de programación anfitrión. En este caso el SQL asume el papel de sublenguaje de datos.

En el caso de hacer un uso embebido del lenguaje podemos utilizar dos técnicas alternativas de programación. En una de ellas, en la que el lenguaje se denomina “SQL Estático”, las sentencias utilizadas no cambian durante la ejecución del programa; es decir, la instrucción SQL se conoce antes de que el programa se ejecute. Los objetos de la base de

[7] La explicación original de Codd fue expuesta en el artículo: “*A Relational Model of Data for Large Shared Data Banks*”.

datos deben de existir cuando se precompilan instrucciones de SQL estático. Este tipo de SQL puede concebirse como una forma compilada del lenguaje SQL. Es una característica que favorece al desempeño. En la otra, donde el lenguaje recibe el nombre de "SQL Dinámico", se produce una modificación total o parcial de las sentencias en el transcurso de la ejecución del programa. La utilización de SQL dinámico permite mayor flexibilidad y mayor complejidad en las sentencias, pero como contra punto obtenemos una eficiencia menor y el uso de técnicas de programación más complejas en el manejo de memoria y variables. Puede concebirse como una forma interpretativa del lenguaje SQL. No es necesario que existan objetos de la base de datos al precompilarse instrucciones de SQL dinámico. El proceso de compilación debe repetirse cada vez que sea ejecutada de nuevo la instrucción. El SQL estático por lo general se utiliza para la generación de programas de transacciones altamente optimizados. El SQL dinámico se usa para la generación de utilerías de programación general de bases de datos y en herramientas frontales o de primer plano de GUI que necesitan crear consultas específicas.

Como ya se mencionó, y como suele ser común en los lenguajes de acceso a bases de datos de alto nivel, el SQL es un lenguaje declarativo. O sea, que especifica que es lo que se quiere y no como conseguirlo, por lo que una sentencia no establece explícitamente un orden de ejecución. El orden de ejecución interno de una sentencia puede afectar gravemente a la eficiencia del DBMS, por lo que se hace necesario que éste lleve a cabo una optimización antes de la ejecución de la misma. Incluye características para definir las estructuras de datos, para la modificación de los mismos y para la especificación de mecanismos de integridad y seguridad. Algunas de sus principales funciones son:

- SQL es un lenguaje de consulta interactivo para consultas específicas de bases de datos. Fue diseñado originalmente como un lenguaje de consulta para el usuario final. Sin embargo, las modernas GUI para bases de datos SQL son ya de uso mucho más intuitivo. Además son muy eficaces para ocultar a los usuarios finales la semántica subyacente de SQL.
- SQL es un lenguaje de programación de base de datos. Se le puede implantar en lenguajes como Pascal, C, Java, entre otros para obtener acceso a datos. SQL es un lenguaje consistente para la programación de datos. Esto eleva la

productividad del programador y contribuye a producir un sistema más flexible y fácil de mantener.

- SQL es un lenguaje de definición y administración de datos. El lenguaje de definición de datos sirve para definir tablas simples, objetos complejos, índices, vistas, restricciones de integridad de referencias y control de seguridad y acceso. Todos los objetos definidos con SQL son automáticamente registrados (y mantenidos) en un diccionario de datos. La estructura y organización de la base de datos SQL se almacena en ella misma.
- SQL es el lenguaje de los servidores de bases de datos en red. Actualmente se le emplea como lenguaje universal para el acceso y la manipulación de todo tipo de datos.
- SQL contribuye a la protección de los datos en entornos de red de usuarios múltiples. Lo hace gracias a que cuenta con características de confiabilidad, como validación de datos, integridad de referencias, retrocesos o anulación de transacciones, candados automáticos, detección y resolución de bloqueo en entornos LAN multiusuarios. También se encarga de la seguridad y control de acceso a objetos de bases de datos.

SQL ofrece un buen número de ventajas a los creadores de sistemas, ya que sirve lo mismo para definir una base de datos que para manipularla. El lenguaje SQL facilita la especificación precisa de requerimientos de productos. Esto favorece la comunicación entre clientes y desarrolladores con los DBA. Proporciona una rica funcionalidad más allá de la simple consulta (o recuperación) de datos. Asume el papel de lenguaje de definición de datos, de lenguaje de definición de vistas y de lenguaje de manipulación de datos. Además permite la concesión y denegación de permisos, la implementación de restricciones de integridad y controles de transacción, y la alteración de esquemas. El lenguaje SQL contiene los siguientes componentes:

- Lenguaje de definición de datos o *Data Definition Language* (DDL). El cual proporciona órdenes para la definición de esquemas de relación, borrado de relaciones, creación de índices y modificación de esquemas de relación.

- Lenguaje interactivo de manipulación de datos o *Data Manipulation Language* (DML). Incluye un lenguaje de consultas, basado tanto en el álgebra relacional como en el cálculo relacional de tuplas. Incluye también órdenes para consultar, insertar, borrar y modificar tuplas de la base de datos.
- DML incorporado. La forma incorporada, incrustada o embebida de SQL se diseñó para el uso sin lenguajes de programación de propósito general, tales como Pascal, C o Java.
- Definición de vistas. El DDL de SQL incluye órdenes para la definición de las diferentes vistas para los diferentes usuarios de la base de datos.
- Autorización. El DDL de SQL incluye también órdenes para la especificación de derechos de acceso relacionales y vistas.
- Integridad. El DDL de SQL incluye órdenes para especificar las ligaduras de integridad (reglas de negocio) que deben satisfacer los datos almacenados en la base de datos. Las actualizaciones que violen las ligaduras de integridad se rechazan.
- Control de transacciones. SQL incluye órdenes para la especificación del comienzo y final de transacciones en las bases de datos.

En una arquitectura Cliente/Servidor centrada en bases de datos, por lo general una aplicación Cliente solicita datos y servicios relacionados con datos (como clasificación y filtración) a un Servidor de bases de datos. El Servidor de bases de datos responde a las solicitudes del Cliente y ofrece acceso protegido a datos compartidos. Con una sola instrucción SQL, una aplicación Cliente puede recuperar y modificar un conjunto de registros de la base de datos del Servidor. El mecanismo de bases de datos de SQL puede filtrar los conjuntos de resultados de la consulta, lo que deriva en considerables ahorros de comunicación de datos.

Un Servidor de bases de datos SQL administra el control y ejecución de comandos SQL. Proporciona las vistas lógica y física de los datos además de generar planes de acceso optimizado para la ejecución de los comandos de SQL. Además la mayoría de los servidores de bases de datos proporcionan características y utilerías de administración del Servidor que facilitan el manejo de los datos. Mantiene tablas dinámicas de catálogos que

contienen información sobre los objetos SQL alojados en él. Administra los aspectos de recuperación, concurrencia, seguridad y consistencia de una base de datos. Esto supone controlar la ejecución de una transacción y anular sus efectos en caso de fallas. Implica asimismo la obtención y entrega de candados durante la ejecución de una transacción y la protección de los objetos de la base de datos contra acceso no autorizado.

Aunque a partir de 1979 han aparecido numerosas implementaciones comerciales de SQL, no se contó con un estándar oficial hasta que el Instituto Estadounidense de Normas Nacionales o ANSI (*American National Standards Institute*) y la Organización de Normas Internacionales o ISO (*International Standards Organization*) emitieron uno en forma conjunta. El primer estándar SQL (SQL1) se le conoce actualmente como SQL-87[8] y su paquete de mejoras, SQL-89. Después surge el SQL-92 o SQL2 (Elmasri y Navathe, 2000) en el cual se amplían características existentes en la anterior publicación y se añaden otras que mejoran la funcionalidad. Ahora, con el avance y cambio en el paradigma de programación, principalmente, surge en 1999 una tercera publicación de este estándar conocido como SQL-99 o SQL3 y es el primero en tener una "orientación a objetos", mejorado en SQL-2003.

Un Servidor actual de SQL es una combinación de SQL estándar en cualquiera de sus publicaciones y extensiones específicas realizadas por el proveedor o constructor del DBMS a que proporcionará el servicio. Por ejemplo, los desarrolladores especializados como Oracle Corporation, tienen sumo interés en extender sus mecanismos de bases de datos para que ejerzan funciones de servidores más allá del modelo de datos relacional, integrando la orientación a objetos. Otros como Microsoft Corporation, se inclinan en adoptar estándares de SQL e instalar extensiones de procedimientos no estandarizadas del DBMS dentro de los NOS y, algunos otros desarrolladores más pequeños se esfuerzan de tener un DBMS que cumpla con todos los estándares SQL. La implementación de distintas extensiones para la obtención y manipulación de datos realizadas por los principales fabricantes de DBMS, así como el uso de distintas publicaciones de SQL dentro de los mismos, hace que el diseño de aplicaciones distribuidas heterogéneas que necesiten acceder

[8] Según la especificación del estándar SQL-99, la versión conocida como SQL-86 cambia su nombre a SQL-87. (ISO/IEC, 2003).

a datos almacenados en los distintos gestores sea muy complicado, especialmente en los tipos de datos que soporta cada DBMS y/o su publicación de SQL utilizada.

Normalmente, SQL está muy bien preparado para tratar datos tradicionales. El modelo relacional de datos define cuidadosamente la semántica de las operaciones sobre las "relaciones" de "tuplas" de "atributos", pero no especifica cuál debe ser el tipo de los datos. SQL tiene los siguientes tipos de datos básicos y publicados en SQL-87 y en SQL-89 (Alarcón, 1994): *SmallInt, Integer, Decimal, Numeric, Float, Real, Double precision,* y *Character*. Con el tiempo y el transcurso de las demás publicaciones, se han añadido nuevos tipos de datos. En SQL-92 (Melton y Simon, 1993) se añadieron los siguientes tipos de datos: *Character varying, National character, National character varying, Bit, Bit varying, Date, Time, Interval* y *Timestamp*. SQL-99 (Melton y Simon, 2001) añade los siguientes tipos de datos: *Character large object, National character large object, Binary large object, Array, Row, Ref* y *Boolean*. Por último, SQL-2003 (ISO/IEC 9075-2:2003) añade a los siguientes: *BigInt, Scope* y *Multiset*.

Los tipos de datos en SQL se usan para representar lo que se denomina "datos tradicionales": números, texto, cadenas de bits y fechas. Los gestores de bases de datos que contenían datos representados exclusivamente por estos tipos han contenido durante muchos años la gran mayoría de los datos de las empresas de todo el mundo (Melton y Eisenberg, 2002). Conforme la tecnología se vuelve cada vez accesible para todos, los usuarios intentan ahora almacenar y administrar colecciones masivas de documentos de texto no estructurado, fotografías, vídeos, música, imágenes de radar, de satélite, datos sismológicos y demás que SQL no ha sido tradicionalmente bueno para estos requerimientos. Aunque muchos fabricantes han mejorado sus productos añadiendo un tipo de datos que se suele denominar como Grandes Objetos Binarios o BLOB (*Binary Large OBjects*) con la intención de permitir a los usuarios almacenar y recuperar datos no tradicionales, el soporte de BLOB se han limitado estrictamente al almacenamiento y recuperación, sin otra semántica aplicable a ellos (como las comparaciones o su uso como criterios para recuperar otros datos). SQL-99 se ha diseñado para soportar a los datos complejos a través de un mecanismo de "tipos definidos por el usuario", en el que se proporcionó la posibilidad de definir directamente en SQL nuevos tipos de datos y sus

operaciones permitiendo así al lenguaje el soporte a los tipos de datos complejos que la industria estaba requiriendo.

II.3. Ingeniería de *Software* y Programación Orientada a Objetos.

Las aplicaciones distribuidas deben de contar con una arquitectura flexible que permita tener sistemas complejos y que puedan ser ampliados, modificados o corregidos a manera de cubrir nuevos requisitos de *software*, adaptarse a nuevos lineamientos o tecnologías de cómputo o simplemente de corregir errores en la programación que disminuyen la calidad del programa. La Ingeniería de *Software* Basada en Componentes o CBSE (*Computer Enginnering Based Components*) es actualmente una de las más prometedoras para incrementar la calidad del *software*, abreviar los tiempos de acceso al mercado y gestionar el continuo incremento de su complejidad.

La Ingeniería de *Software* es un conjunto de métodos, técnicas y herramientas que controlan el proceso integral del desarrollo de *software* y suministra las bases para construir *software* de calidad de forma eficiente en los plazos adecuados; vea Figura 2.15 (Pressman, 2002). La Ingeniería de *Software* es el análisis, el diseño, la construcción, la verificación y la gestión de entidades técnicas. El trabajo asociado puede ser dividido en tres fases genéricas, con independencia del área de aplicación, tamaño o complejidad del proyecto. Cada fase se encuentra con una o varias cuestiones de las destacadas anteriormente.

La fase de análisis se centra sobre el “qué”. Es decir, durante la definición, el “qué” desarrolla el *software* intenta identificar qué información ha de ser procesada, qué función y rendimiento se desea, qué comportamiento del sistema, qué interfaces van a ser establecidas, qué restricciones de diseño existen, y qué criterios de validación se necesitan para definir un sistema correcto. Por tanto, ha de especificarse los requisitos clave del sistema y del *software*. Aunque los métodos aplicados durante la fase de definición variarán dependiendo del paradigma de Ingeniería de *Software* (o combinación de paradigmas) que se aplique, de alguna manera tendrán lugar tres etapas principales: ingeniería de sistemas o de información, planificación del proyecto de *software* y el análisis de requisitos.

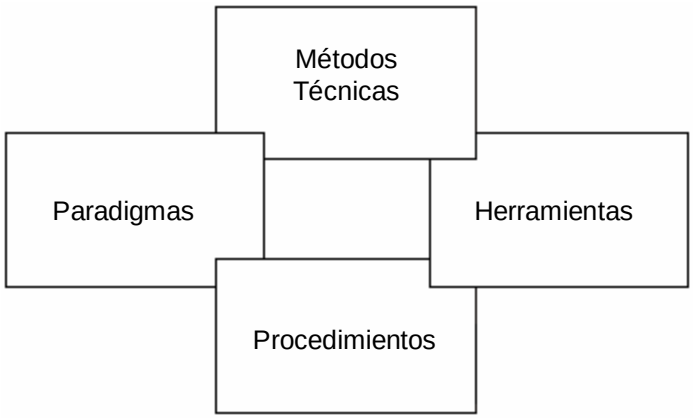

Figura 2.15. Elementos de la Ingeniería del *Software*.

La fase de diseño o definición se centra en el "cómo". Es decir, durante el desarrollo se intenta definir cómo han de diseñarse las estructuras de datos, cómo ha de implementarse los procedimientos, cómo han de caracterizarse las interfaces, cómo ha de traducirse el diseño en un lenguaje de programación y cómo ha de realizarse las pruebas. Los métodos aplicados durante la fase de desarrollo variarán, aunque las tres tareas específicas técnicas deberían ocurrir siempre: el diseño de *software*, generación de código y la prueba del *software*.

La fase de desarrollo del *software* se puede caracterizar como un bucle de resolución de problemas en el que se encuentran cuatro capas distintas: estado, definición de problemas, desarrollo técnico e integración de soluciones; vea Figura 2.16. El estado actual representa la situación actual de sucesos; la definición de problemas identifica el problema específico a resolverse; el desarrollo técnico resuelve el problema a través de la aplicación de alguna tecnología y la integración de soluciones ofrece los resultados (por ejemplo: documentos, programas, datos, etcétera) a los que solicitan la solución en primer lugar.

Para el mundo actual de la programación y para el desarrollo de sistemas distribuidos muy flexibles se utiliza el paradigma de Programación Orientada a Objetos u POO (*Programming Object Oriented*). Este estilo en la programación incorpora características del modelo mostrado en la Figura 2.16, es evolutivo y exige un enfoque iterativo para la creación de *software*. El Modelo de Desarrollo Basado en Componentes configura aplicaciones desde componentes preparados de *software*; las tecnologías de

objeto proporcionan el marco de trabajo técnico para un modelo de proceso basado en componentes para la Ingeniería del *Software*. Los sistemas orientados a objetos tienden a evolucionar con el tiempo. Por esto, un modelo evolutivo de proceso acoplado con un enfoque que fomenta el ensamblaje (reutilización) de componentes es el mejor paradigma para Ingeniería del *Software* Orientada a Objetos. El proceso OO se mueve a través de una espiral evolutiva que comienza con la comunicación con el usuario. Es aquí en donde se define el dominio del problema. La planificación y el análisis de riesgo establecen una base para el plan del proyecto OO. El trabajo técnico asociado con la Ingeniería de *Software* OO sigue el camino iterativo mostrado en la Figura 2.17.

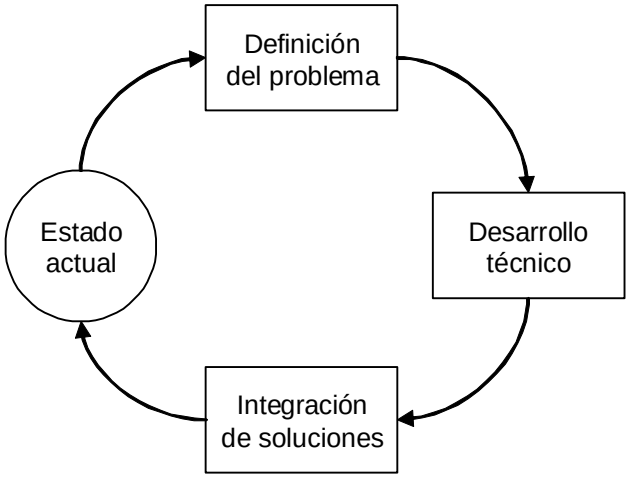

Figura 2.16. Las fases en un ciclo de resolución de problemas.

La Ingeniería de *Software* OO hace hincapié en la reutilización. Por lo tanto, los componentes se buscan en una biblioteca antes de construirse. Cuando un componente no puede encontrarse en la biblioteca, el desarrollador de *software* aplica análisis orientado a objetos, diseño orientado a objetos, programación orientada a objetos y pruebas orientadas a objetos para crear los componentes. El nuevo componente se coloca en la biblioteca de tal manera que pueda ser reutilizada en un futuro. La visión orientada a objetos demanda un enfoque evolutivo de la Ingeniería del *Software*.

La fase de mantenimiento y pruebas se centra en el “cambio” que va asociado a la corrección de errores, a las adaptaciones requeridas a medida que evoluciona el entorno del

software y a cambios debidos a las mejoras producidas por los requisitos cambiantes del Cliente.

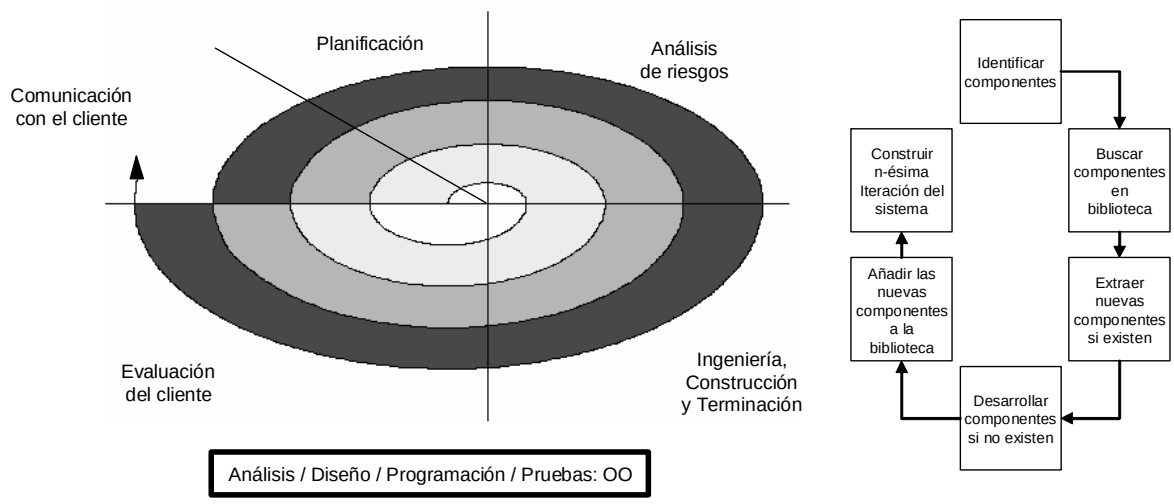

Figura 2.17. El modelo de proceso orientado a objetos.

II.3.1. El Lenguaje Unificado de Modelado.

Al final de la década de los noventa, surgió el Lenguaje Unificado de Modelado o UML (*Unified Modeling Language*) como un estándar para la recopilación de toda la información que compone al modelo que se obtiene del análisis, el diseño y la programación. Junto a UML fue desarrollado el Proceso Unificado de Desarrollo de *Software* por sus creadores Grady Booch, Ivar Jacobson y James Rumbaugh, que representa un número de modelos de desarrollo y diseño basados en componentes que han sido propuestos utilizando UML. El proceso unificado define los componentes que se utilizarán para construir el sistema y las interfaces que conectarán los componentes. Utilizando una combinación del desarrollo incremental e iterativo, el proceso unificado define la función del sistema aplicando un enfoque basado en escenarios. Entonces acopla la función con un marco de trabajo arquitectónico que identifica la forma que tomará el *software*.

UML es un lenguaje gráfico para visualizar, especificar, construir y documentar los componentes o artefactos de un sistema con gran cantidad de *software*. Proporciona una forma estándar de escribir las especificaciones de un sistema, cubriendo tanto las cosas

conceptuales, tales como procesos del negocio y funciones del sistema, como las cosas concretas, tales como las clases escritas en un lenguaje de programación específico, esquemas de bases de datos y componentes de *software* reutilizables.

En UML (Booch, et. al. 1999), un sistema viene representado por cinco vistas diferentes perspectivas. Cada vista se representa mediante un conjunto de diagramas y estas son:

- Vista del usuario. Representa al sistema (producto) desde la perspectiva de los usuarios (llamados actores). El caso de uso es el enfoque elegido para modelar esta vista. Esta importante representación del análisis describe un escenario de uso desde la perspectiva del usuario final.
- Vista estructural. Los datos y la funcionalidad se muestran desde dentro del sistema, es decir, modela la estructura estática (clases, objetos y relaciones).
- Vista del comportamiento. Esta parte del modelo del análisis representa los aspectos dinámicos o de comportamiento del sistema. También muestra las interacciones o colaboraciones entre los diversos elementos estructurales descritos en las vistas anteriores.
- Vista de implementación. Los aspectos estructurales y de comportamiento se representan aquí tal cual y como van a ser implementados.
- Vista del entorno. Aspectos estructurales y de comportamiento en el que el sistema a implementar se representa.

Cada una de las vistas anteriormente citadas está representada a través del uso de una notación gráfica que se engloban en los "Diagramas UML". Los diagramas con los medios para ver un sistema desde diversas perspectivas. Como ningún sistema puede ser comprendido completamente desde un único punto de vista, UML define varios diagramas que permiten centrarse en diferentes aspectos del sistema independientemente. Éstos se encuentran divididos en dos grupos:

- Parte estática de un sistema: Diagrama de clases, de objetos, de componentes y de despliegue.

- Parte dinámica de un sistema: Diagrama de casos de uso, de secuencia, de colaboración, de estados y de actividades.

Un DIAGRAMA DE CLASES presenta un conjunto de clases, interfaces y colaboraciones, y las relaciones entre ellas. Los diagramas de clases son los diagramas más comunes en el modelado de sistemas orientados a objetos. Los diagramas de clase se utilizan para describir la vista de diseño estática de un sistema. Los diagramas de clases que incluyen clases activas se utilizan para cubrir la vista de procesos estática de un sistema. Vea la Figura 2.18.

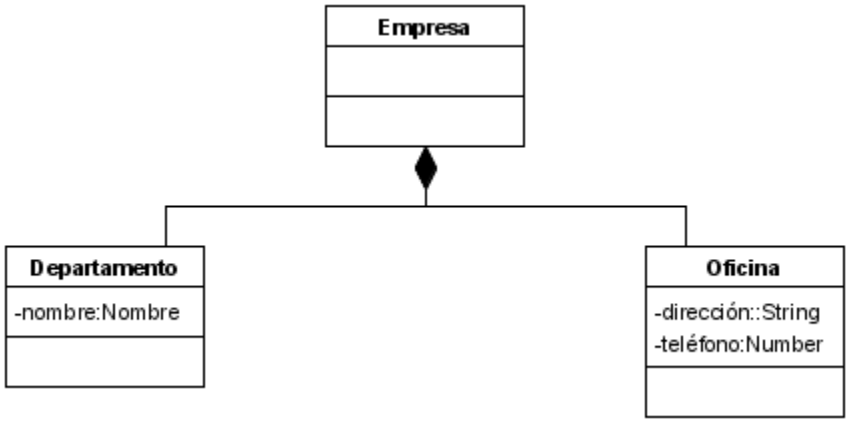

Figura 2.18. Diagrama de clases.

Un DIAGRAMA DE OBJETOS representa un conjunto de objetos y sus relaciones. Se utilizan para describir estructuras de datos, instantáneas de las instancias de los elementos encontrados en los diagramas de clases. Los diagramas de objetos cubren la vista de diseño estática o la vista de procesos estática de un sistema al igual que los diagramas de clases, pero desde la perspectiva de casos reales o prototípicos. Vea la Figura 2.19.

Un DIAGRAMA DE COMPONENTES muestra un conjunto de componentes y sus relaciones. Los diagramas de componentes se utilizan para describir la vista de implementación estática de un sistema. Los diagramas de componentes se relacionan con los diagramas de clases en que un componente normalmente se corresponde con una o más clases, interfaces o colaboraciones. Vea Figura 2.20.

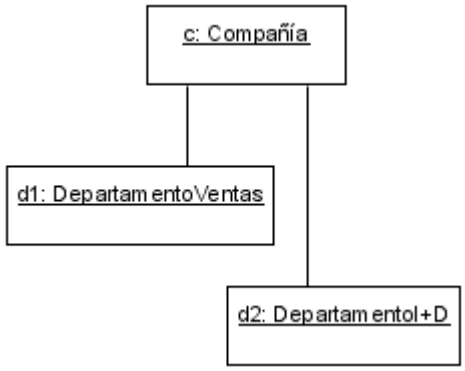

Figura 2.19. Diagrama de objetos.

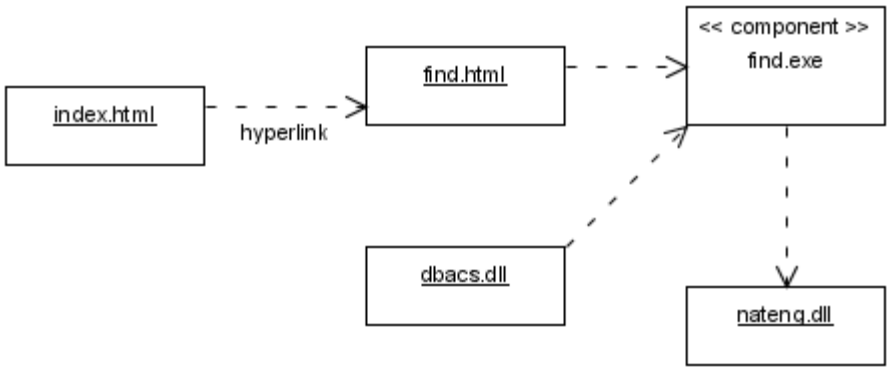

Figura 2.20. Diagrama de componentes.

Un DIAGRAMA DE DESPLIEGUE muestra un conjunto de nodos y sus relaciones. Los diagramas de despliegue se utilizan para describir la vista de despliegue estática de una arquitectura. Los diagramas de despliegue se relacionan con los diagramas de componentes en que un nodo normalmente incluye uno o más componentes. Vea Figura 2.21.

Un DIAGRAMA DE CASOS DE USO representa un conjunto de casos de uso y actores (un tipo especial de clases) y sus relaciones. Los diagramas de este tipo se utilizan para describir la vista de casos de uso estática de un sistema. Estos diagramas son especialmente importantes para organizar y modelar el comportamiento de un sistema. Vea la Figura 2.22.

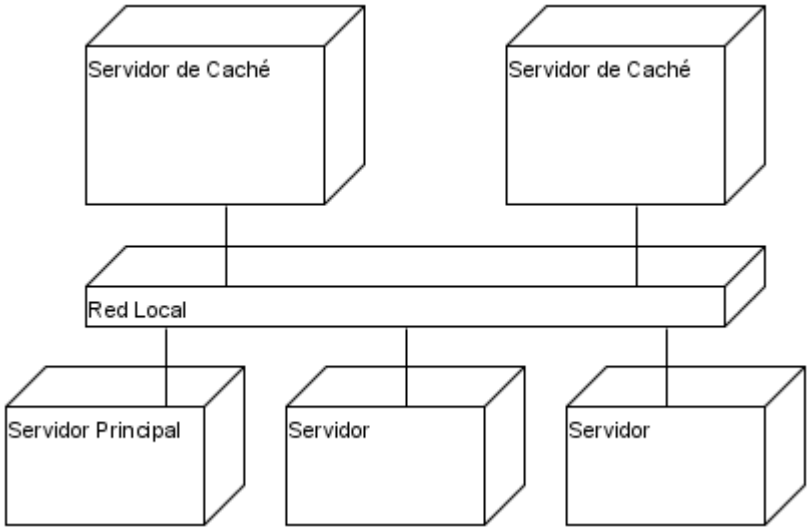

Figura 2.21. Diagrama de despliegue.

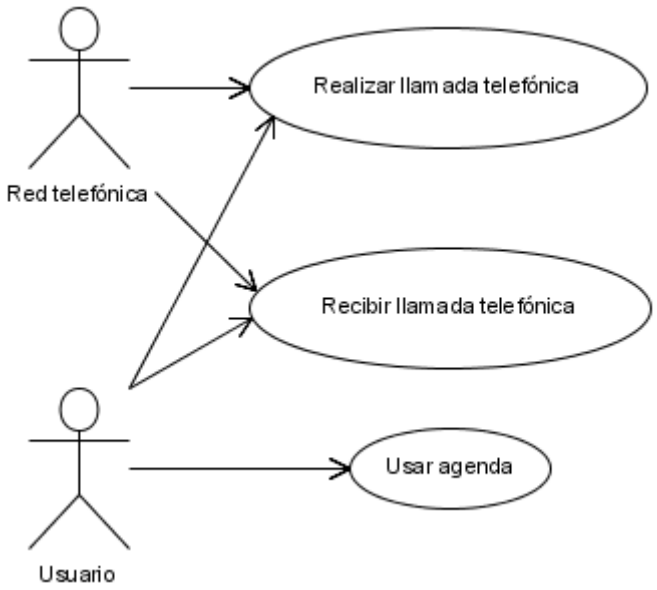

Figura 2.22. Diagrama de casos de uso.

Un DIAGRAMA DE SECUENCIA es un diagrama de interacción que resalta la ordenación temporal de los mensajes. Un diagrama de secuencia presenta un conjunto de objetos y los mensajes enviados y recibidos por ellos. Los objetos suelen ser instancias con nombre o anónimas de clases, pero también pueden representar instancias de otros

elementos, tales como colaboraciones, componentes y nodos. Los diagramas de este tipo se utilizan para describir la vista dinámica de un sistema. Vea la Figura 2.23.

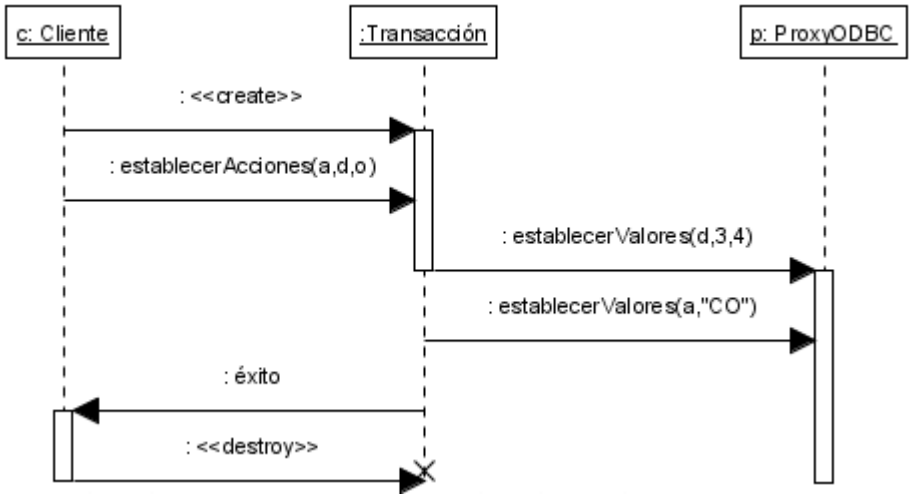

Figura 2.23. Diagrama de secuencia.

Un DIAGRAMA DE COLABORACIÓN es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Un diagrama de colaboración muestra un conjunto de objetos, enlaces entre esos objetos y mensajes enviados y recibidos por esos objetos. Los objetos normalmente son instancias con nombre o anónimas de clases, pero también pueden representar instancias de otros elementos, como colaboraciones, componentes y nodos. Los diagramas de colaboración se utilizan para describir la vista dinámica de un sistema. Vea la Figura 2.24.

Un DIAGRAMA DE ESTADOS representa una máquina de estados, constituida por estados, transacciones, eventos y actividades. Los diagramas de estados se utilizan para describir la vista dinámica de un sistema. Son especialmente importantes para modelar el comportamiento de una interfaz, una clase o una colaboración. Los diagramas de estados resaltan el comportamiento dirigido por eventos de un objeto, lo que es especialmente útil para modelar sistemas reactivos. Vea Figura 2.25.

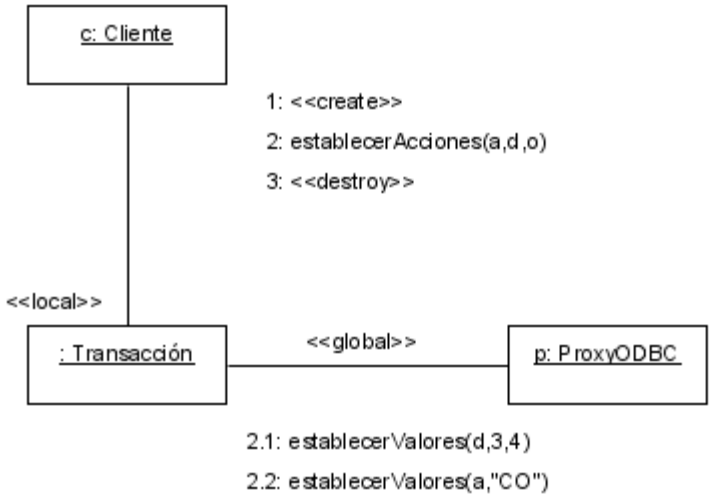

Figura 2.24. Diagrama de colaboración.

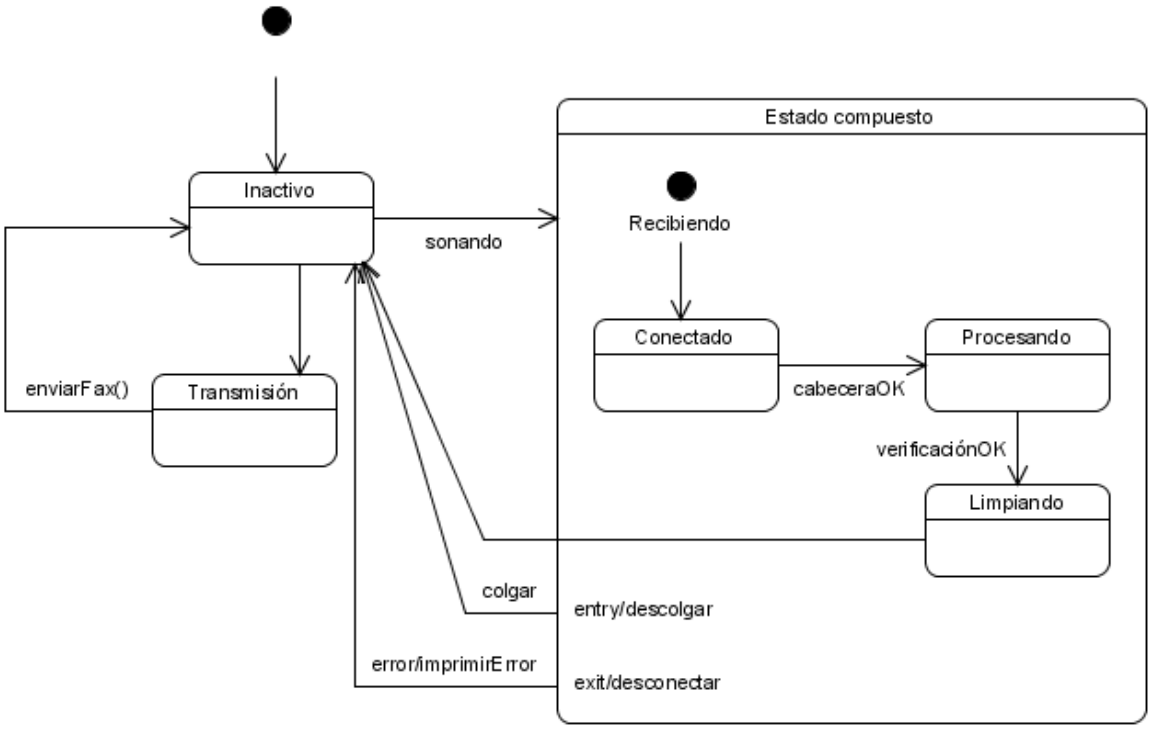

Figura 2.25. Diagrama de estados.

Un DIAGRAMA DE ACTIVIDADES muestra el flujo de actividades de un sistema. Una actividad muestra un conjunto de actividades, el flujo secuencial o ramificado de

actividades, y los objetos que actúan y sobre los que se actúa. Los diagramas de actividades se utilizan para ilustrar la vista dinámica de un sistema. Además, estos diagramas son especialmente importantes para modelar la función de un sistema, así como para resaltar el flujo de control entre objetos. Vea Figura 2.26.

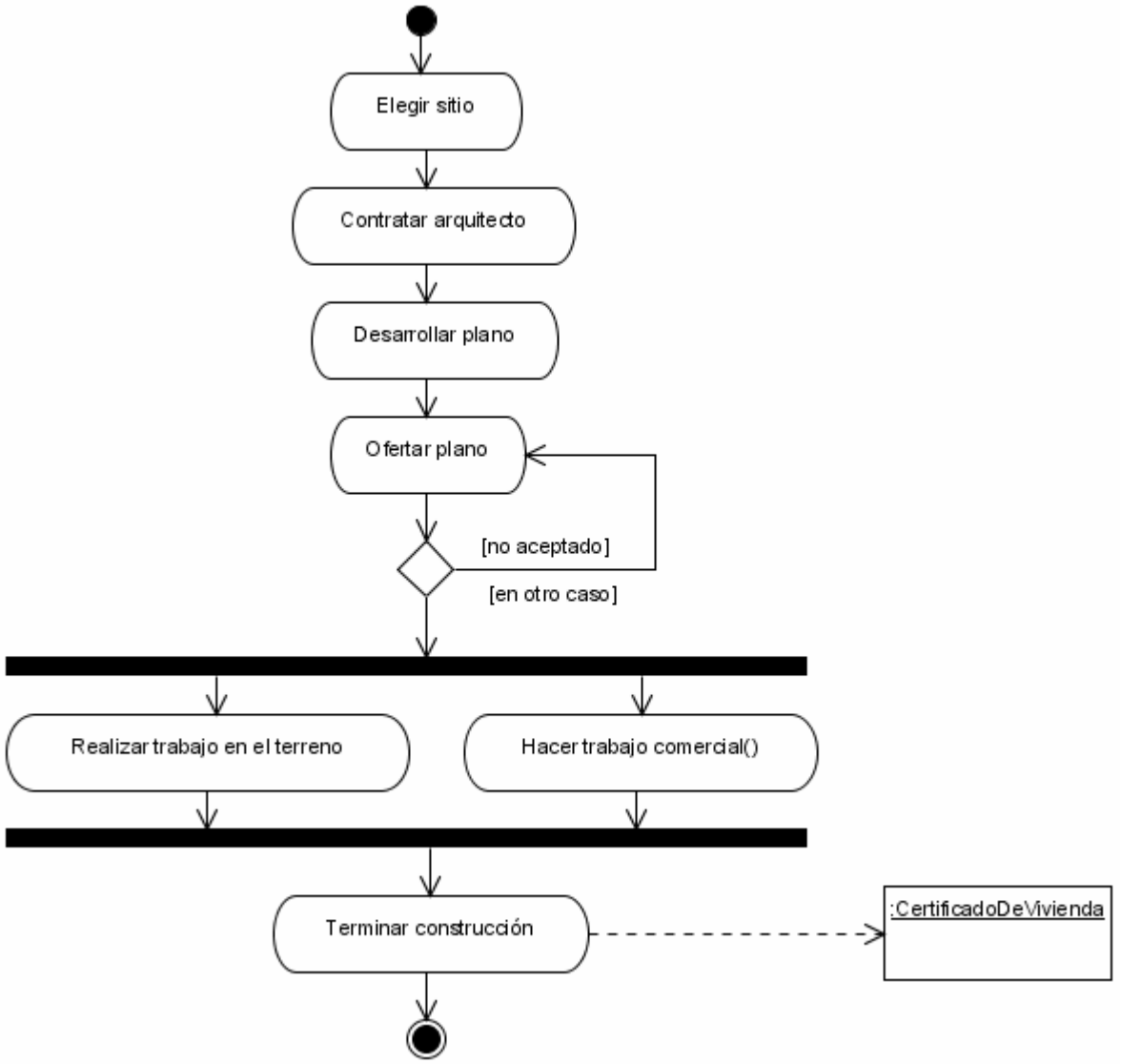

Figura 2.26. Diagrama de actividades.

En general, el modelo de análisis de UML, se centra en las vistas de usuario y estructural. El modelo de diseño de UML se dirige más a las vistas de comportamiento y del entorno y aunque se ideó para el modelado de sistemas computacionales, también es lo

suficientemente expresivo para modelar sistemas que no son *software*, como flujos de trabajo en una organización.

II.3.2. El lenguaje de programación Java y bases relacionales de datos.

La Programación Orientada a Objetos expresa un programa como un conjunto de estos objetos, que se comunican entre ellos para realizar tareas. Esto difiere de los lenguajes por procedimientos tradicionales en los que los datos y los procedimientos están separados y sin relación. Estos métodos están pensados para hacer los programas y módulos más fáciles de escribir, mantener y reutilizar. La programación orientada a objetos anima al programador a pensar en los programas principalmente en términos de tipos de datos y en operaciones específicas a esos tipos de datos. Los lenguajes por procedimiento animan al programador a pensar sobre todo en términos de procedimientos, y en los datos que esos procedimientos manejan. Los programadores que emplean lenguajes por procedimientos escriben funciones y después les pasan datos. Los programadores que emplean lenguajes orientados a objetos definen objetos con datos y métodos para después enviar mensajes a los objetos diciendo que realicen esos métodos en sí mismos (Ceballos, 2000).

Java que es una plataforma de *software* desarrollada inicialmente por James Gosling en la empresa Sun Microsystems y que su principal característica es que es multiplataforma de tal manera que los programas creados en este lenguaje puedan ejecutarse sin cambios en diferentes tipos de arquitecturas y dispositivos computacionales. La portabilidad significa que una aplicación pueda ser ejecutada en cualquier computadora que ejecute cualquier sistema operativo permitiendo que la computadora interprete de manera adecuada cada una de las instrucciones, los valores dados e incluso los datos a compartir entre los usuarios. Este ideal recae en gran medida en el lenguaje de programación utilizado para codificar las aplicaciones informáticas. Para lograr esto, Java incluye dos elementos: un compilador y un intérprete. El compilador produce un código de *bytes* que se almacena en un archivo para ser ejecutado por el intérprete llamado Máquina Virtual de Java o JVM (*Java Virtual Machine*).

El lenguaje de programación Java en sí es muy pequeño, sin embargo la forma en que está diseñado le permite ser extensible y crecer en sus capacidades y dependiendo de

las aplicaciones futuras que quieran desarrollarse con él. Al ser pensado para Internet, Java se ha construido con extensas capacidades de interconexión utilizando la pila de protocolos TCP/IP, además, existen rutinas para acceder e interactuar con el Protocolo de Transferencia de Hipertexto o http (*Hypertext Transfer Protocol*), lo cual permite a los programadores obtener información a través de la red con tanta facilidad como a los archivos locales. Java en sí no es distribuido, sino que proporciona las librerías y las herramientas para que los programas puedan ser distribuidos, es decir, que se ejecuten en varías computadoras y plataformas interactuando.

Las Interfaces para la Programación de Aplicaciones o API (*Application Program Interfaces*) son elementos que permiten extender la funcionalidad de cualquier aplicación *software* y Java no es la excepción. Este lenguaje es capaz de poder ser utilizado para programar diversos tipos de aplicaciones para atacar a igual número de problemas a resolver. A través de las API, Java crece en su funcionalidad. Java, al ser un lenguaje de programación orientado a objetos, con portabilidad en la ejecución de sus programas y al ser muy flexible permite programar de una manera muy adecuada, aplicaciones que requieren acceso a sistemas de bases de datos relacional, ya sean locales o distribuidas, a través de una API llamada Conectividad en Java para Base de Datos o JDBC (*Java Data Base Connectivity*) y por medio de consultas al DBMS utilizando el lenguaje SQL de acceso a datos. Para que una aplicación Java pueda hacer operaciones en una base de datos ha de realizar primero una conexión con ella, que se establece a través de un controlador o *driver*, que convierte el lenguaje de alto nivel a sentencias de base de datos. Las tres acciones principales que realizará JDBC son las de establecer la conexión a una base de datos, ya sea remota o no; enviar sentencias SQL a esa base de datos y, en tercer lugar, procesar los resultados obtenidos de la base de datos. Las bases de datos son las estructuras más utilizadas en computación, ya que son el núcleo de sistemas muy complejos y de los cuales depende toda la información y el futuro de una organización. Los datos contenidos en estos sistemas se valúan en grandes cantidades de dinero. JDBC está diseñado teniendo en mente la comunicación con bases de datos. Especifica una serie de clases y métodos para permitir a cualquier programa Java una forma homogénea o común de acceso a sistemas de bases de datos. Este acceso se realiza a través de *drivers*, que son los que implementan la funcionalidad especificada en JDBC.

Existe un intento por lograr esto por parte de Microsoft con una API llamada Conectividad Abierta con Bases de datos u ODBC (*Open Data Base Connectivity*), sin embargo, éste se encuentra diseñado en C y al no ser un lenguaje que puede ejecutarse en cualquier microprocesador sin hacerle muchos cambios, haría que las aplicaciones de cualquier lenguaje incluso Java perdiesen portabilidad, lo que va en contra de las especificaciones de este lenguaje. Además, ODBC tiene el inconveniente de que se ha de instalar en cada computadora; al contrario que los *drivers* JDBC, que al estar escritos en Java son automáticamente instalables, portátiles y seguros.

La conectividad de bases de datos se basa en sentencias SQL, que a través de JDBC, permiten realizar la conexión a la base de datos, realizar consultas y recibir los resultados. JDBC permite la actualización de múltiples registros con un solo comando o acceder a múltiples servidores de bases de datos dentro de una transacción simple, permite reutilizar las conexiones a la base de datos de modo que no se requiere realizar una nueva conexión cada vez que se quiera realizar alguna acción en ella (Froufe, 2000).

El API JDBC soporta dos modelos diferentes de acceso a bases de datos, los modelos de dos y tres capas. El modelo de dos capas se basa en que la conexión entre la aplicación Java se conecta directamente a la base de datos, vea Figura 2.27.

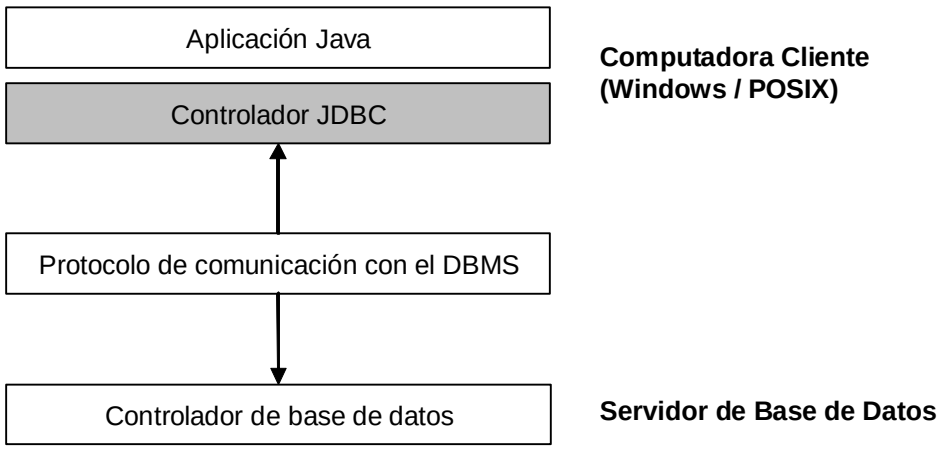

Figura 2.27. Modelo de dos capas JDBC.

Esto significa que el *driver* JDBC específico para conectarse con la base de datos, debe de residir en el sistema local. La base de datos puede estar en cualquier otra máquina y

se accede a ella mediante la red. Ésta es la configuración típica Cliente/Servidor, el programa cliente envía instrucciones SQL a la base de datos, ésta las procesa y envía los resultados de vuelta a la aplicación.

En el modelo de tres capas, las instrucciones son enviadas a una capa intermedia entre Cliente y Servidor, que es la que se encarga de enviar las sentencias SQL a la base de datos y recoger el resultado desde la base de datos, vea la Figura 2.28. En este caso, el usuario no tiene contacto directo, ni siquiera a través de la red, con la máquina en donde reside la base de datos.

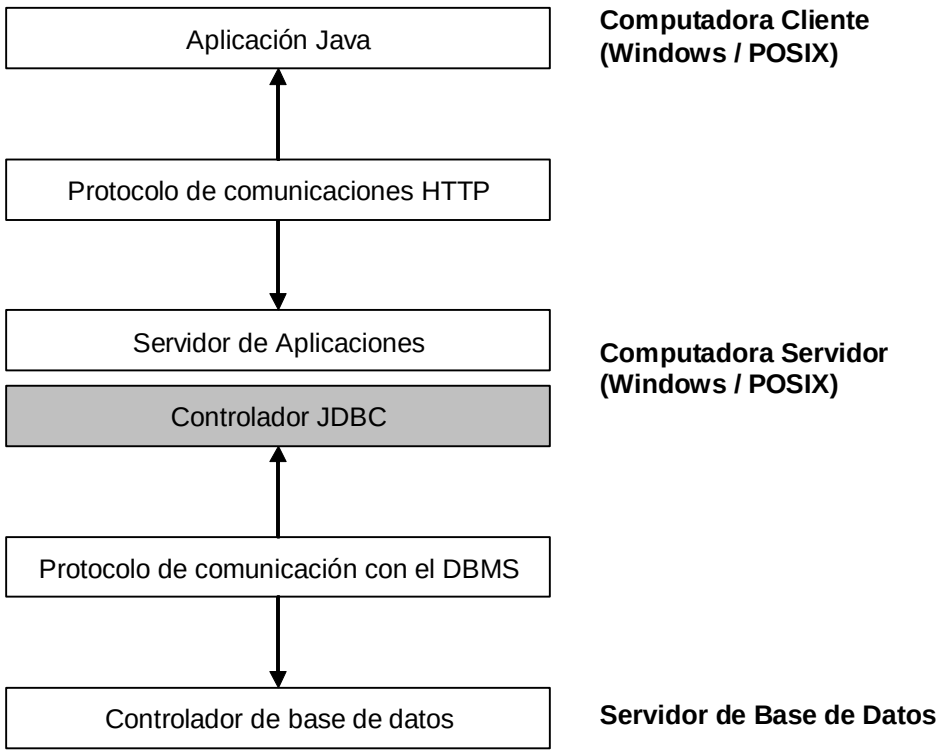

Figura 2.28. Modelo de tres capas JDBC.

Este modelo presenta la ventaja de que el nivel intermedio mantiene en todo momento el control del tipo de operaciones que se realizan contra la base de datos, y además, está la ventaja adicional de que los *drivers* JDBC no tienen que residir en la máquina cliente, lo cual libera al usuario de la instalación de cualquier tipo de *driver*.

Para conectarse a bases de datos individuales, JDBC necesita un *driver* específico para cada una de ellas (Melton y Eisenberg, 2002). Hay cuatro tipos de *drivers* y son:

- El Puente JDBC-ODBC es el *driver* que proporciona acceso a bases de datos desde JDBC a través de uno o más *drivers* ODBC, representando una forma de aprovechar todo lo que ya exista, sin necesidad de realizar configuraciones adicionales. Este *driver* proporciona una forma excelente de acercarse a JDBC, y puede resultar útil cuando ya hay *drivers* ODBC instalados en las computadoras cliente, como suele ser el caso de las máquinas que están ejecutando aplicaciones Windows. Vea la Figura 2.29.

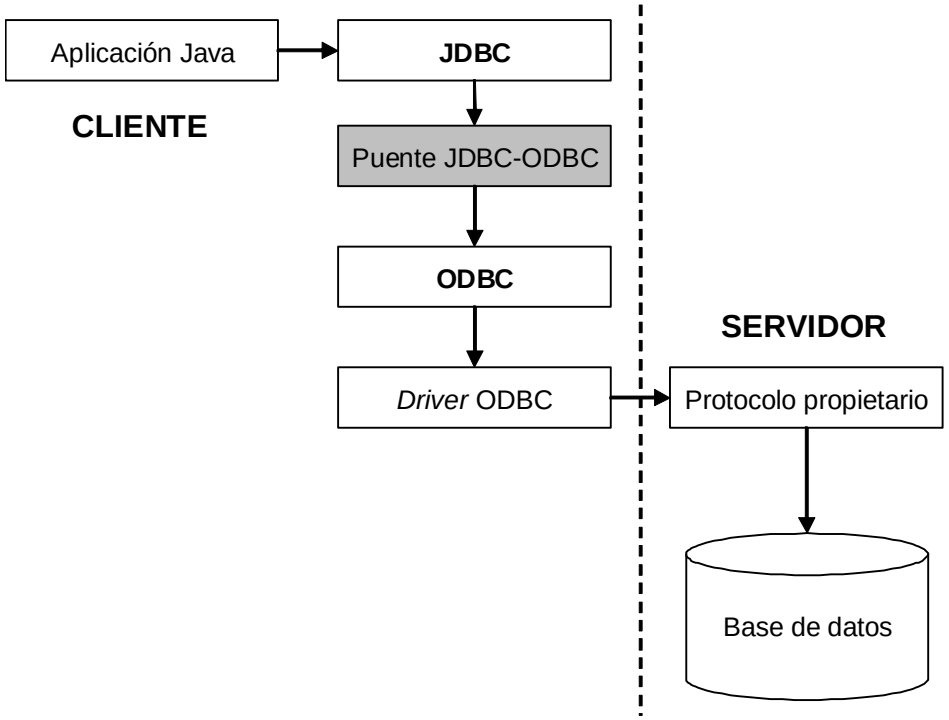

Figura 2.29. Puente JDBC-ODBC.

Este *driver* es inadecuado cuando se trata de aplicaciones de cierta entidad, porque el rendimiento se resiente enormemente al tener que realizarse la conversión de las transacciones de JDBC a ODBC. Además este *driver* no soporta todas las características de

Java y la programación se ve limitada por la funcionalidad que ofrezca el *driver* ODBC que esté utilizando.

Normalmente el *driver* ODBC se carga de forma local a través de una librería, lo cual impide el acceso a datos a través de la red. Por ello, los *drivers* JDBC de este tipo son adecuados solamente en tiempo de desarrollo, ya que para utilizarlos en entornos de red hay que recurrir a otra API llamada Invocación de Métodos Remotos o RMI (*Remote Method Invocation*) utilizada mucho en aplicaciones distribuidas, que replica una conexión local en una base de datos remota, incluyendo una nueva capa que hace disminuir el rendimiento de la aplicación.

- El controlador Java/Binario es una variación del anterior controlador y en este se realiza una comunicación directa con la librería que proporciona el fabricante. En rendimiento es mejor que el del Puente JDBC-ODBC porque contiene la parte de código independiente del sistema ya compilada para el acceso a la base de datos de que se trate, cuyo *driver* debe estar cargado en cada máquina cliente. Vea la Figura 2.30.

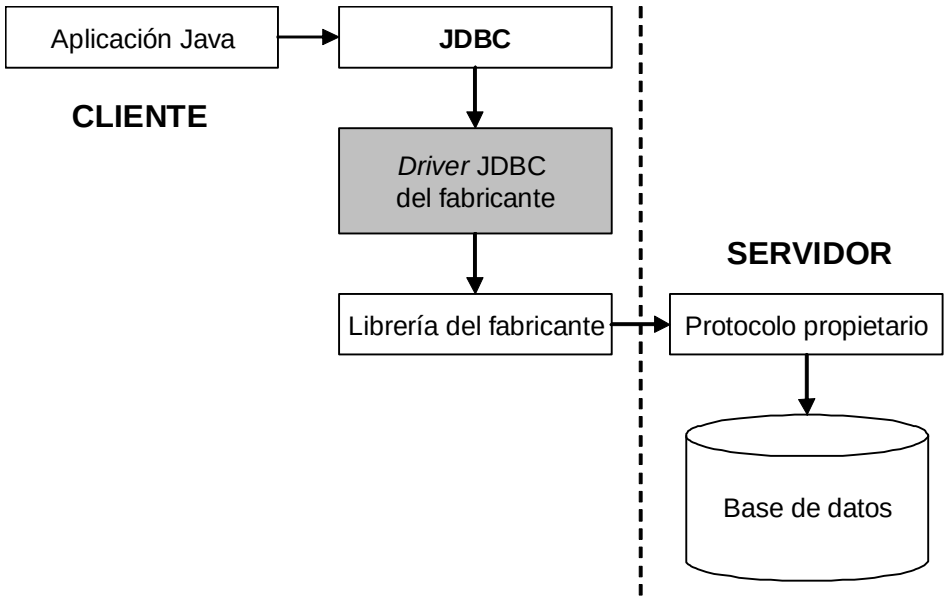

Figura 2.30. Controladores Java/Binario.

Por ejemplo, el sistema de base de datos Oracle es el que proporciona este tipo de *driver* para acceso a sus bases de datos. Este tipo de *drivers* elimina la parte ODBC de los *drivers* tipo JDBC-ODBC y proporciona rendimientos aceptables en entornos Intranet.

- El tercer tipo de *driver* es el 100% Java/Protocolo nativo y es sin duda el tipo de controlador más utilizado y se le conoce también como "Controlador de Protocolo de Red", ya que está realizado totalmente en Java y convierte directamente las peticiones JDBC del cliente en peticiones de red contra el servidor. Esta conversión se realiza en el cliente. Vea la Figura 2.31.

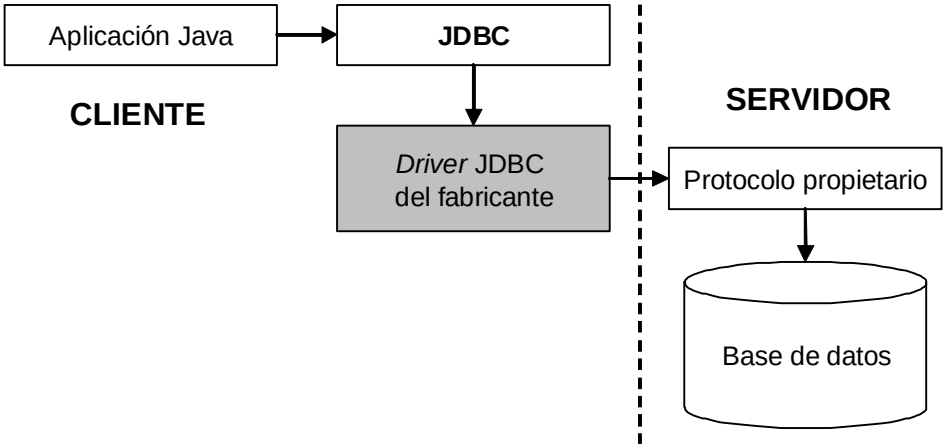

Figura 2.31. Controlador 100% Java/Protocolo nativo.

Este *driver* puede manejar múltiples clientes Java conectados a múltiples bases de datos. Además, como puede conectarse a diferentes direcciones de res, permite el desarrollo de aplicaciones Cliente/Servidor que sigan el modelo de tres capas. El *driver* se ejecuta como un demonio o *daemon*, así como un servicio sobre el sistema Servidor.

- El último tipo de controlador es el 100% Java/Protocolo independiente y es el más flexible de los cuatro. Convierte las llamadas JDBC directamente en llamadas nativas hacia el sistema DBMS de que se trate. Esta conversión se realiza en el servidor, el cliente puede utilizar un *driver* JDBC genérico. Vea la Figura 2.32.

La ventaja de este tipo de controladores es que no se necesita nada especial en el cliente, sino que todo el control se ejerce desde el servidor, lo cual hace muy simple el cambio de un servidor a otro. Sin embargo, estos *drivers*, al estar desarrollados por los fabricantes de DBMS, están optimizados para proporcionar toda la funcionalidad que cada uno de estos fabricantes ofrece, pero no tienen en cuenta las ventajas o características que pueda ofrecer el OS sobre el que se ejecutan.

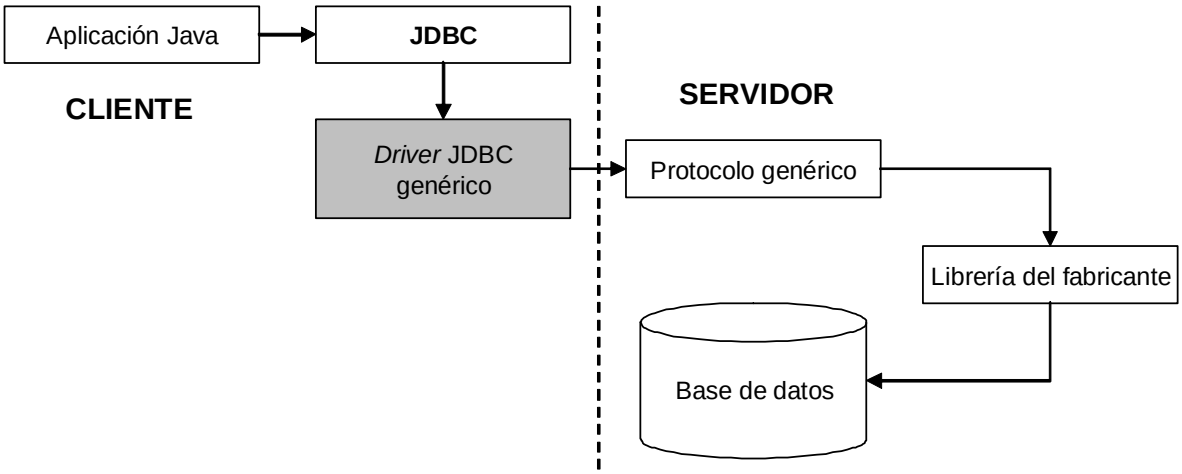

Figura 2.32. Controlador 100% Java/Protocolo independiente.

Existe otro factor muy importante para el manejo de bases de datos en Java. Los programas Java que intenten acceder a los datos almacenados deben ser capaces de manejarlos de una manera adecuada porque si no se pueden presentar problemas de redondeo de valores, sobrecarga (el superar el máximo valor posible que puede almacenar una variable) o pérdida de precisión si se eligen tipos de datos con menos capacidad de almacenamiento durante la conversión que se tenga que realizar. Aunque SQL y Java están diseñados para ser lo más transportables posible entre distintas plataformas, han usado soluciones muy diferentes para conseguir esta finalidad.

El Lenguaje de Base de Datos SQL principalmente está diseñado para gestionar grandes cantidades de datos tradicionales persistentes que tienen asociada una semántica inherente. En contraste, la misión de Java es gestionar objetos y lo hace proporcionando la definición de clases que permiten dar soporte a una extensa variedad de posibles

aplicaciones por ser un lenguaje de programación se propósito general. La diferencia entre estas dos herramientas hace que al utilizarlos juntos, muchos de los tipos de datos de SQL no tienen un tipo de datos equivalente en Java, es decir, existen diferencias en cuanto a la capacidad de almacenamiento de cada tipo de dato para guardar los valores que se le proporcionen. Utilizar los tipos de datos de manera inadecuada haría que el almacenar valores de una variable de Java y a ésta dentro de una base de datos diera como resultado una mala interpretación o pérdida de significado de los valores por parte de SQL puesto que este lenguaje y Java no manejan de igual manera a un carácter o a un número. Incluso cuestiones como el juego de de caracteres del OS o de idioma juega un papel importante.

Java inicialmente fue diseñado para que cada elemento dentro de él fuera utilizado como un objeto. Los tipos de datos no escapan a esta característica puesto que existen tipos de datos "de referencia" que utilizan a los tipos de datos "primitivos" para almacenar valores y que además pueden realizarse ciertas operaciones en ellos al tener métodos para accederlos (Gosling, et. al., 1996).

El Cuadro 2.1 muestra las equivalencias de los tipos de datos primitivos Java con sus correspondientes tipos en SQL, es decir, los tipos de datos que utiliza un motor de base de datos para almacenar los valores asignados a las variables de Java. El Cuadro 2.2 muestra cada tipo de datos SQL (hasta la versión 2003) junto con el nombre del tipo primitivo de Java que le corresponde.

Por último, el Cuadro 2.3 muestra la equivalencia de los tipos de datos SQL con los tipos de datos por referencia de Java.

2.1. Equivalencias a los tipos de datos SQL desde Java.

TIPOS DE DATOS PRIMITIVOS DE JAVA	TIPOS DE DATOS PREDEFINIDOS DE SQL
byte	Ninguno
short	SMALLINT
int	INTEGER
long	Ninguno
float	REAL
double	DOUBLE PRECISION
char	NATIONAL CHARACTER (Dimensión: 1)
boolean	BOOLEAN

2.2. Equivalencias a los tipos de datos primitivos Java desde SQL.

TIPOS DE DATOS PREDEFINIDOS DE SQL	TIPOS DE DATOS PRIMITIVOS DE JAVA
SMALLINT	short
INTEGER INT	int
DECIMAL DEC	Ninguno
NUMERIC	
FLOAT	double
REAL	float
DOUBLE PRECISION	double
CHARACTER CHAR	Ninguno
NATIONAL CHARACTER NATIONAL CHAR NCHAR	Únicamente soporta los valores de tamaño de 1
NATIONAL CHARACTER VARYING NATIONAL CHAR VARYING NCHAR VARYING	Ninguno
BIT	
BIT VARYING	
DATE	
TIME	
TIMESTAMP	
INTERVAL	
CHARACTER LARGE OBJECT CHAR LARGE OBJECT CLOB	
NATIONAL CHARACTER LARGE OBJECT NCHAR LARGE OBJECT NCLOB	
BINARY LARGE OBJECT BLOB	
BOOLEAN	boolean
ARRAY	Vectores con dimensiones y declarados con un tipo de datos en especial
ROW	Ninguno
REF	
BIGINT	
SCOPE	
MULTISET	

Cuadro 2.3. Equivalencias a los tipos de datos por referencia Java desde SQL.

TIPOS DE DATOS PREDEFINIDOS DE SQL	TIPOS DE DATOS POR REFERENCIA DE JAVA
SMALLINT	java.lang.Integer
INTEGER INT	
DECIMAL DEC	java.math.BigDecimal
NUMERIC	
FLOAT	java.lang.Double
DOUBLE PRECISION	
REAL	java.lang.Float
CHARACTER CHAR	java.lang.String
NATIONAL CHARACTER NATIONAL CHAR NCHAR	
NATIONAL CHARACTER VARYING NATIONAL CHAR VARYING NCHAR VARYING	
BIT	java.lang.Boolean
BOOLEAN	
BIT VARYING	Ninguno
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
INTERVAL	
CHARACTER LARGE OBJECT CHAR LARGE OBJECT CLOB	java.sql.Clob
NATIONAL CHARACTER LARGE OBJECT NCHAR LARGE OBJECT NCLOB	
BINARY LARGE OBJECT BLOB	java.sql.Blob
ARRAY	java.sql.Array
ROW	Ninguno
REF	java.sql.Ref
BIGINT	java.lang.Long
SCOPE	Ninguno
MULTISET	

III. METODOLOGÍA

III.1. Especificación de una Interfaz Común SQL.

III.1.1. Generalidades.

La “Interfaz Común SQL” es un componente de *software* llamado *middleware* que se utiliza dentro del contexto de un Sistema Distribuido y sobre el cual funciona un tipo de aplicación Cliente/Servidor llamado Sistema de Base de Datos en Federación. Cabe recordar que este sistema tiene un cierto grado de heterogeneidad y que se encuentra delimitada por la implementación de distintas tecnologías de DBMS que permiten, a su vez, un nivel de autonomía local en los sitios de red que la componen. También, y para dar la ilusión de una base de datos única, se requiere que una base de datos distribuida utilice el método del “mínimo común denominador” (Orfali, et. al., 1998) en su diseño. Éste propone que la creación de una federación, el cuál es un híbrido entre los sistemas centralizados con los distribuidos, se encuentre integrada por medio de un solo esquema conceptual para que el usuario final de una aplicación Cliente perciba la mayoría de las 12 consideraciones del Principio Fundamental de una Base de Datos Distribuida (Date, 2001) que dice: “*Ante el usuario, un sistema distribuido debe lucir exactamente igual que en un sistema que no lo es*”. Se puede indicar entonces que la heterogeneidad de un Sistema de Base de Datos en Federación viene dada, tanto por el *software* que le da forma a las aplicaciones como por el *hardware* sobre el cual estas funcionan.

Gracias al protocolo de comunicaciones TCP/IP que actualmente se utiliza para conectar diferentes tipos de computadoras en todo el mundo con una gran variedad de medios físicos de conexión, los límites anteriores en cuanto a *hardware* “prácticamente” desaparecen, puesto que podemos enviar y recibir información a cualquier parte conocida desde nuestros hogares. “Prácticamente”, porque aunque existan computadoras con diferentes microprocesadores integrados que hacen imposible que se comuniquen entre sí, el *software* ayuda a que estas limitantes sean disminuidas a cierto grado gracias a los diferentes Sistemas Operativos que existen y que controlan al *hardware*, así como a los

lenguajes de programación como Java que permite crear aplicaciones que puedan ser ejecutadas sobre cualquier computadora no importando la cantidad de bits que un microprocesador pueda manejar por segundo o la marca del fabricante de los mismos.

La presente investigación concluye en la creación de un prototipo llamado "Interfaz Común SQL" que permitió la comunicación (intercambio de datos) entre una aplicación Cliente con diferentes aplicaciones Servidor representadas por tres DBMS instalados en computadoras que se dividieron la ejecución de dos sistemas operativos distintos y conectadas en una red TCP/IP. Todo esto logrando un equilibrio en la funcionalidad global del sistema, que se desea esté repartida en los tres niveles de la arquitectura de aplicaciones Cliente/Servidor (Figura 3.1).

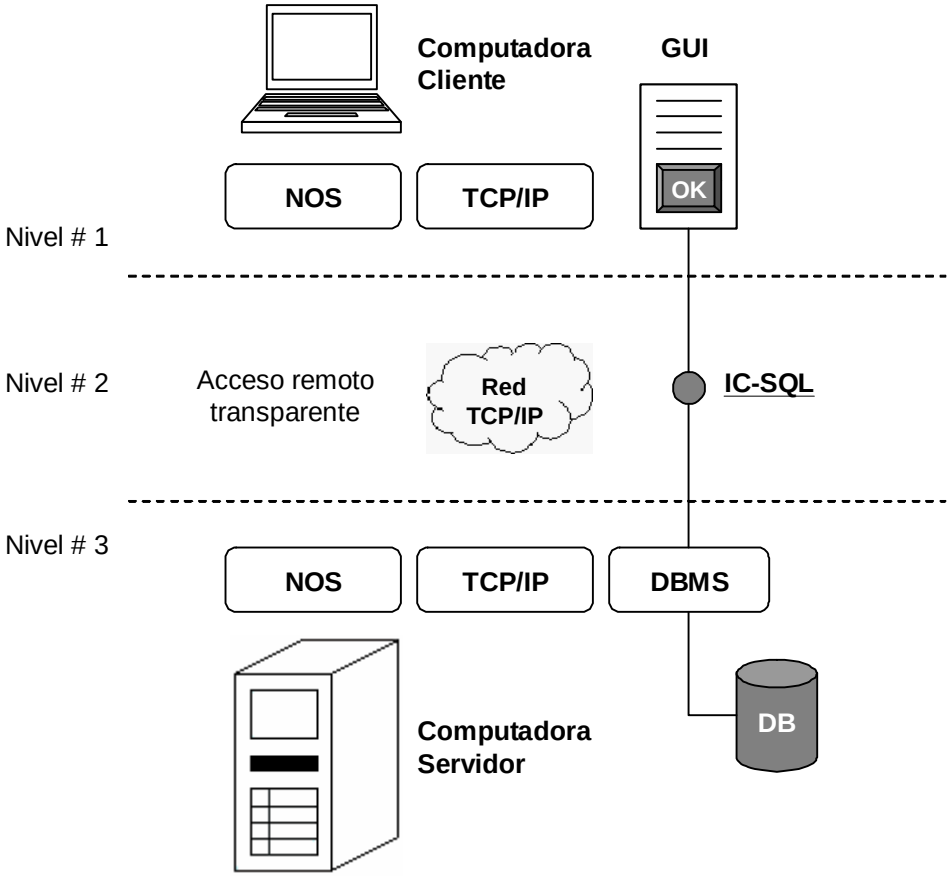

Figura 3.1. Arquitectura Cliente/Servidor e Interfaz Común SQL.

III.1.2. Descripción del Campo de acción experimental.

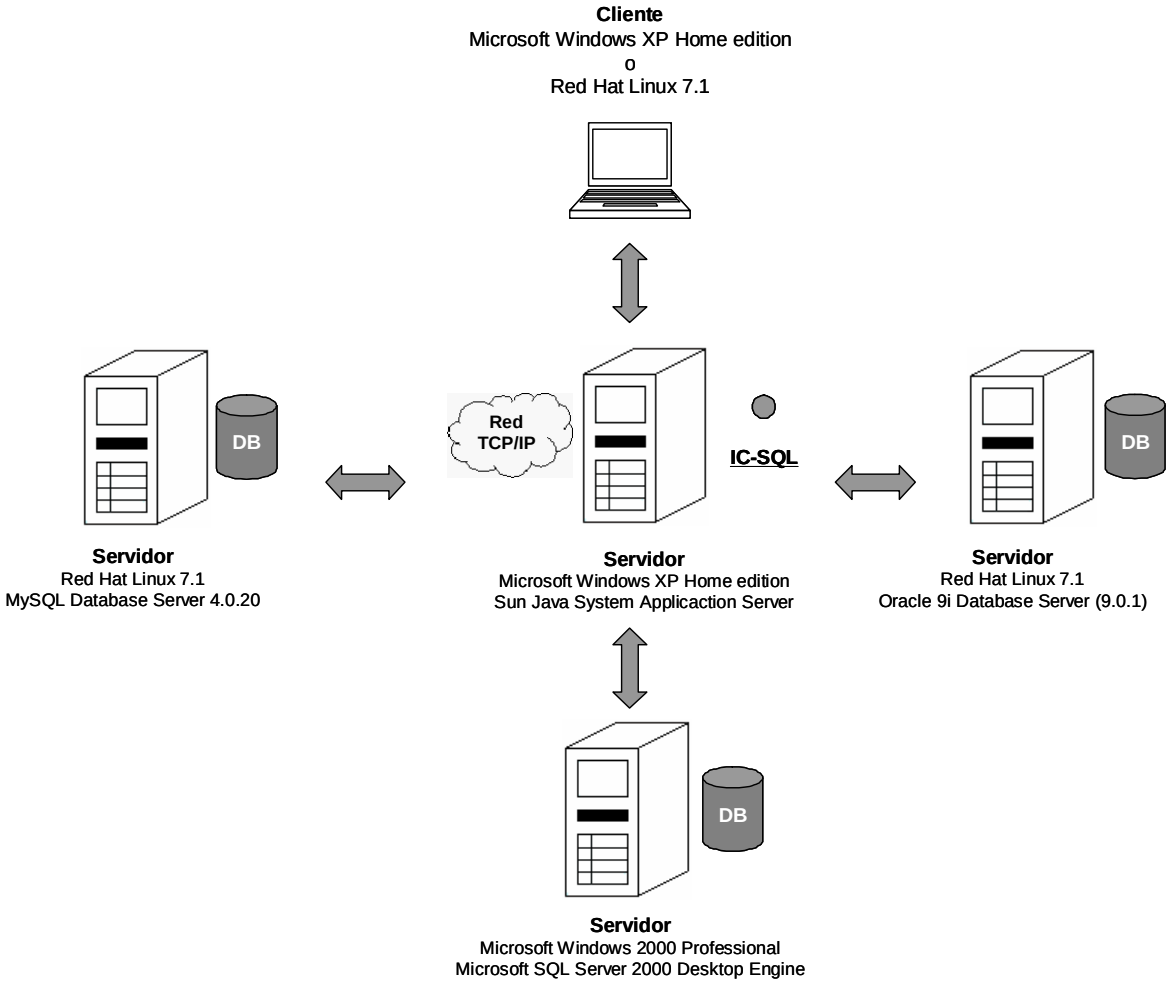

Figura 3.2. Ambiente heterogéneo y distribuido de base de datos.

Este componente se implementó en un campo de acción experimental (Figura anterior, 3.2) que dio forma a una base de datos distribuida, utilizándose las siguientes herramientas:

- Cinco computadoras, tanto físicas como virtuales, que formaron una red de computadoras conectadas a través del protocolo TCP/IP y direccionamiento estático. Para las computadoras virtuales se eligió el programa Microsoft Virtual PC versión 2004.

- Dos sistemas operativos diferentes: Microsoft Windows en dos versiones: 2000 (edición Professional) y XP (Home edition), además de GNU/Linux, disponible a través de la distribución Red Hat versión 7.0.1 (Seawolf).
- Tres diferentes Sistemas de Administración de Bases de Datos para la creación de una base de datos distribuida: Microsoft SQL Server versión 2000 (edición Desktop Edition), MySQL Database Server versión 4.0.20 y Oracle 9i Database Server versión 9.0.1.
- Un servidor Web y de Aplicaciones para tener una representación individual de un *middleware* en donde se ejecuta la "Interfaz Común SQL" y se proporciona el servicio de una aplicación práctica accesible por las terminales cliente. Se eligió el Sun Java System Application Server versión 8.2 (Platform Edition).

Normalmente, una base de datos distribuida debe de tener la siguiente arquitectura de referencia para comprender de manera lógica la organización de este tipo de sistemas, vea la Figura 3.3, y que determina el que una aplicación Cliente crea que se utiliza un sistema centralizado de base de datos.

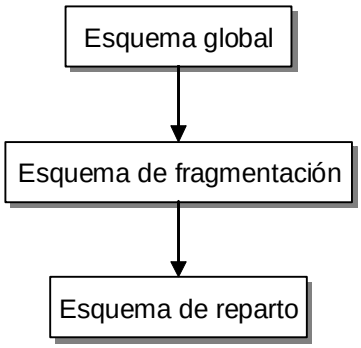

Figura 3.3. Arquitectura de referencia de una base de datos distribuida.

El esquema global es el modelo de datos lógico que se creó para diseñar y describir la base de datos de manera centralizada proveniente del modelo conceptual que se creó en la fase de diseño conceptual (Elmasri y Navathe, 2000). La Figura 3.4, por ejemplo, muestra una vista centralizada puesto que es la forma en que cómo los usuarios ven al

sistema; ayuda a la independencia de ubicación y de fragmentación, entre otras consideraciones que fueron revisadas en el capítulo II.2.1. En este caso, se trata del modelo de datos relacional puesto que los DBMS elegidos para ser la plataforma de una aplicación distribuida y heterogénea manejan este tipo de modelo por su afinidad al Lenguaje de Base de Datos SQL, que está orientado a los modelos relacionales de base de datos.

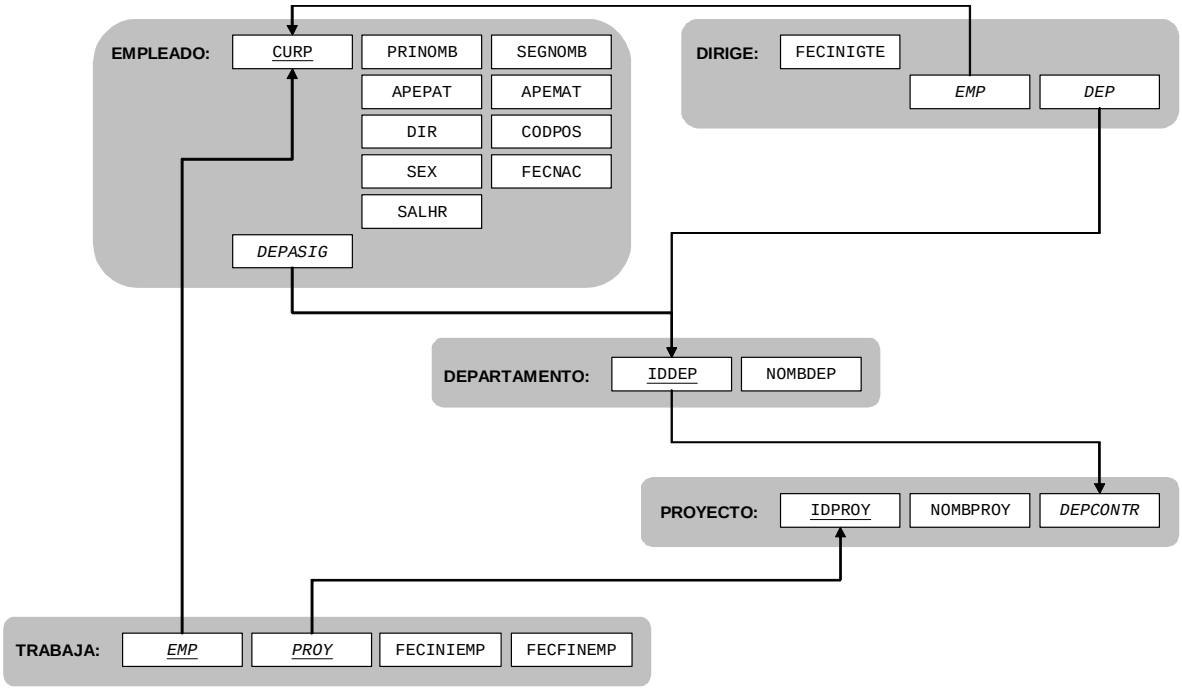

Figura 3.4. Esquema global de una base de datos distribuida.

En (Elmasri y Navathe, 2000), el Esquema de Fragmentación es una definición del conjunto de fragmentos que incluye todas las columnas y filas del modelo relacional o Esquema Global, que satisface la condición de que la base de datos completa se puede reconstruir a partir de los fragmentos mediante una secuencia de operaciones Reunión ($*$) y Unión ($\cup$) del álgebra relacional.

El esquema de fragmentación horizontal que se muestra a continuación determina que el esquema global de la Figura 3.4 está dividido en varios segmentos horizontales, dependiendo de una condición semántica en común. La condición que sirve para realizar la fragmentación es responsabilidad del diseñador de la base de datos acorde con los requerimientos del usuario final.

$$DEP1 \leftarrow \sigma_{IDDEP=1}(DEPARTAMENTO)$$
$$DEP2 \leftarrow \sigma_{IDDEP=2}(DEPARTAMENTO)$$
$$DEP3 \leftarrow \sigma_{IDDEP=3}(DEPARTAMENTO)$$

$$DEPARTAMENTO \leftarrow (DEP1 \cup DEP2) \cup DEP3$$

$$EMP1 \leftarrow \sigma_{DEPASIG=1}(EMPLEADO)$$
$$EMP2 \leftarrow \sigma_{DEPASIG=2}(EMPLEADO)$$
$$EMP3 \leftarrow \sigma_{DEPASIG=3}(EMPLEADO)$$

$$EMPLEADO \leftarrow (EMP1 \cup EMP2) \cup EMP3$$

$$PROY1 \leftarrow \sigma_{DEPCONTR=1}(PROYECTO)$$
$$PROY2 \leftarrow \sigma_{DEPCONTR=2}(PROYECTO)$$
$$PROY3 \leftarrow \sigma_{DEPCONTR=3}(PROYECTO)$$

$$PROYECTO \leftarrow (PROY1 \cup PROY2) \cup PROY3$$

$$DIR1 \leftarrow \sigma_{DEP=1}(DIRIGE)$$
$$DIR2 \leftarrow \sigma_{DEP=2}(DIRIGE)$$
$$DIR3 \leftarrow \sigma_{DEP=3}(DIRIGE)$$

$$DIRIGE \leftarrow (DIR1 \cup DIR2) \cup DIR3$$

$$TRAB1 \leftarrow \pi_{EMP,PROY,FECINIEMP,FECFINEMP}(\pi_{CURP}(\sigma_{DEPASIG=1}(EMPLEADO)) \underset{CURP=EMP}{\bowtie} TRABAJA)$$
$$TRAB2 \leftarrow \pi_{EMP,PROY,FECINIEMP,FECFINEMP}(\pi_{CURP}(\sigma_{DEPASIG=2}(EMPLEADO)) \underset{CURP=EMP}{\bowtie} TRABAJA)$$
$$TRAB3 \leftarrow \pi_{EMP,PROY,FECINIEMP,FECFINEMP}(\pi_{CURP}(\sigma_{DEPASIG=3}(EMPLEADO)) \underset{CURP=EMP}{\bowtie} TRABAJA)$$

$$TRABAJA \leftarrow (TRAB1 \cup TRAB2) \cup TRAB3$$

La fragmentación vertical hace que los atributos de una relación sean los que determinen la división de la misma, sin embargo, sigue siendo una cuestión de semántica.

A cada fragmento se le debe trasladar la clave primaria de la relación para que exista un vínculo entre ellos. El esquema de fragmentación vertical de esas relaciones es el siguiente:

$$EMP1PER \leftarrow \pi_{CURP,PRINOMB,SEGNOMB,APEPAT,APEMAT,DIR,CODPOS,SEX,FECNAC}(EMP1)$$
$$EMP1PRO \leftarrow \pi_{CURP,SALHR,DEPASIG}(EMP1)$$

$$EMP1 \leftarrow EMP1PER * EMP1PRO$$

$$EMP2PER \leftarrow \pi_{CURP,PRINOMB,SEGNOMB,APEPAT,APEMAT,DIR,CODPOS,SEX,FECNAC}(EMP2)$$
$$EMP2PRO \leftarrow \pi_{CURP,SALHR,DEPASIG}(EMP2)$$

$$EMP2 \leftarrow EMP2PER * EMP2PRO$$

$$EMP3PER \leftarrow \pi_{CURP,PRINOMB,SEGNOMB,APEPAT,APEMAT,DIR,CODPOS,SEX,FECNAC}(EMP3)$$
$$EMP3PRO \leftarrow \pi_{CURP,SALHR,DEPASIG}(EMP3)$$

$$EMP3 \leftarrow EMP3PER * EMP3PRO$$

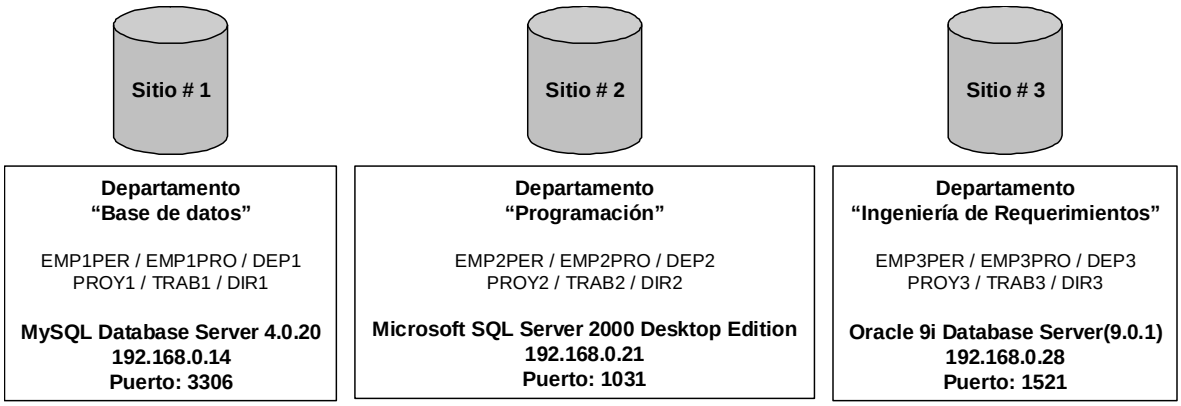

Figura 3.5. Esquema de reparto.

Para el campo de acción experimental, el Esquema de Reparto de la Figura anterior (3.5), muestra la ubicación de cada uno de los fragmentos verticales y horizontales de la base de datos distribuida y que funciona sobre los distintos DBMS que la sostienen.

Además se muestra también el número de dirección del protocolo IP para identificar cada uno de los servidores de bases de datos relacional.

III.2. Estructura Estática y Construcciones de Implementación.

En (Orfali, et. al. 1998) se plantea una cuestión que trata de resolver una problemática que actualmente y en ciertas aplicaciones empresariales se plantea: ¿Cómo accede una aplicación Cliente a la información almacenada dentro de una base de datos con SQL y que se encuentra sobre Servidores de base de datos que fueron construidas por diferentes fabricantes de *software*?.

Una de las alternativas proporcionadas es la creación de un *middleware* capaz de utilizar e igualar los diferentes dialectos y extensiones de SQL que cada DBMS posee, los protocolos de comunicaciones para mensajes de red y de las API específicas de acceso a cada motor relacional participante, comprendiendo también la transmisión de una solicitud de la aplicación Cliente que invoca un servicio por la red y, la respuesta resultante. Sin embargo, la Ingeniería de *Software* determina que lo mejor de las aplicaciones Cliente/Servidor es que se puede repartir de una manera adecuada la cantidad de procesos que estos dos elementos ejecutan para no sobrecargar a los servidores y que éstos puedan seguir proporcionando sus servicios de manera continua, por lo que el diseño de un Sistema Distribuido de Base de Datos, como el que se muestra en el campo de acción experimental, debe también de involucrar en parte a la aplicación Cliente que trabaja en conjunto con el Servidor a través del *software* intermedio.

La creación de esta especificación conlleva a la estandarización de los mecanismos de obtención y manipulación de datos de los motores de base de datos seleccionados uniéndolos bajo un esquema conceptual único. También involucra que la "Interfaz Común SQL" debe de conocer y manejar la mayor parte de las instrucciones del lenguaje de base de datos SQL para que el Cliente pueda interactuar con el Servidor y así intercambiar datos e información. Sin embargo, las instrucciones que se acompañan a cada motor relacional de los DBMS son modificadas por los fabricantes con el único fin de incorporar funcionalidad extra a los mismos para posicionarse dentro del mercado

informático. Desafortunadamente estas alteraciones a lo que se considera un estándar internacional afecta en gran medida la finalidad de la creación del mismo.

Con este propósito la "Interfaz Común SQL" utiliza únicamente el estándar definido por ANSI e ISO en 1992 de las instrucciones de obtención y/o manipulación de datos y que se considera la "versión más apegada al modelo relacional de base de datos". Se seleccionaron las siguientes instrucciones (Figura 3.6):

- La instrucción "SELECT" es la operación más compleja dentro de SQL. Permite la recuperación de datos almacenados en las tablas gestionadas por el DBMS y que juntos dan forma a la información que solicita un usuario.
- La instrucción "INSERT" se utiliza para el ingreso de nuevos datos en cada una de las tablas a las que se haga referencia además de especificar a que columnas se les agregarán esos valores.
- La instrucción "UPDATE" funciona de manera similar a la anterior. La diferencia es que esta se utiliza para la actualización de ciertos datos ya almacenados dentro de la tabla o para añadir nuevos en las columnas de las filas que no tengan valores. Posiblemente se hayan introducido mal algunos datos que requieran una corrección para obtener información confiable o para adecuar lo existente a las nuevas condiciones de la organización.
- Por último, la instrucción "DELETE" se utiliza para eliminar datos que ya no corresponden con la actualidad de la organización.

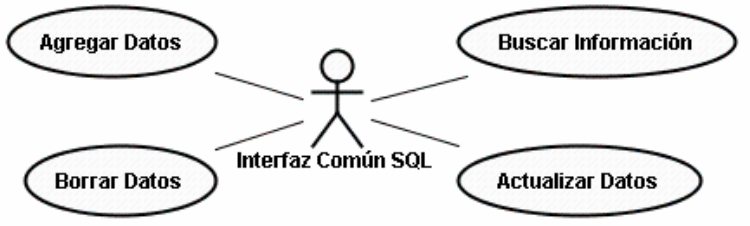

Figura 3.6. Interfaz Común SQL y operaciones con base de datos.

Una muestra de lo anteriormente expuesto fue el uso de tres diferentes motores relacionales de base de datos. Microsoft SQL Server es una herramienta de administración

de bases de datos relaciones basadas en el lenguaje SQL por medio de un conjunto de extensiones de éste llamado “Transact SQL” que incluye un entorno gráfico de administración, que permite el uso de comandos DLL y DML gráficamente, proporcionando un entorno de trabajo en modo Cliente/Servidor donde la información y datos se alojan en el Servidor y las terminales o clientes de la red sólo acceden a ésta, además de permitir la administración de datos de otros servidores similares y, manejando de una forma adecuada la seguridad en cuanto al acceso a las bases de datos. Oracle Database Server es otro DBMS al que se le considera como el “sistema de bases de datos más completo que existe” destacando su soporte de transacciones, gran estabilidad, seguridad, escalabilidad, además es multiplataforma y utiliza un juego de extensiones al lenguaje SQL con el nombre de “PL/SQL”. Su mayor problema es su precio. MySQL Database Server es una de las bases de datos más populares desarrolladas bajo la filosofía del código abierto. Desarrollada en Suecia, posee la característica de poder ser ejecutada en una gran cantidad de plataformas y sistemas, posee diferentes opciones de almacenamiento según si se desea velocidad en las operaciones o el mayor número de transacciones disponibles, conectividad segura, replicación y búsqueda e indexación de campos de texto en un entorno altamente configurable.

La Figura anterior (3.6) muestra, finalmente, desde el punto de vista del usuario final a la “Interfaz Común SQL”, de esa manera es percibido por él, como un elemento que le permite realizar operaciones de obtención y/o manipulación de los datos contenidos dentro de una base de datos.

La “Interfaz Común SQL” proporciona la interacción entre una aplicación Cliente con los distintos Servidores, que integran la base de datos distribuida, por medio de distintos módulos interrelacionados y elementos externos que configuran a este componente como se muestra en la Figura 3.7 y que se describen a continuación:

- Control de ejecución. Este módulo es el punto de unión entre las aplicaciones Cliente, el *middleware* y los Servidores de la base de datos distribuida. Es el inicio y el fin de la interacción entre estos componentes.
- Generador de consultas. La tarea principal de este elemento es la unión de las sentencias seleccionadas del lenguaje SQL con los datos de entrada de las

aplicaciones Cliente para ser ejecutadas por los Servidores de la base de datos distribuida.

- Comunicaciones: Se encarga de establecer y terminar las conexiones a los Servidores de la base de datos distribuida. Además permite la ejecución de las instrucciones de obtención y/o manipulación de datos generadas por la "Interfaz Común SQL" dentro de los motores relacionales involucrados.
- Esquema conceptual: Es un elemento de configuración del *middleware* que contiene el modelo lógico conceptual e integrado de la base de datos distribuida; es la manera en cómo los usuarios finales y las aplicaciones Clientes acceden a la misma.
- Esquema de fragmentación: Contiene el esquema de fragmentación horizontal, vertical o ambos que le da forma a la base de datos distribuida representada en el esquema conceptual.
- Esquema de reparto: Determina la ubicación de los fragmentos de cada tabla de la base de datos conceptual en los sitios de red involucrados y que sostienen a la misma.
- Lista de servidores DBMS: Elemento externo en donde se encuentran los parámetros de conexión de la "Interfaz Común SQL" con los Servidores de la base de datos distribuida.
- Controladores de base de datos: Contiene la colección de *drivers* de conexión a las bases de datos involucradas que le permite a la aplicación, la ejecución de las instrucciones de obtención y/o manipulación de datos requeridas por el usuario final.

La Figura 3.7 muestra la vista estructural de la "Interfaz Común SQL", es la manera en cómo interactúan los distintos elementos que la componen para cumplir con las peticiones enviadas desde la aplicación Cliente que desea acceder a los Servidores de la base de datos distribuida.

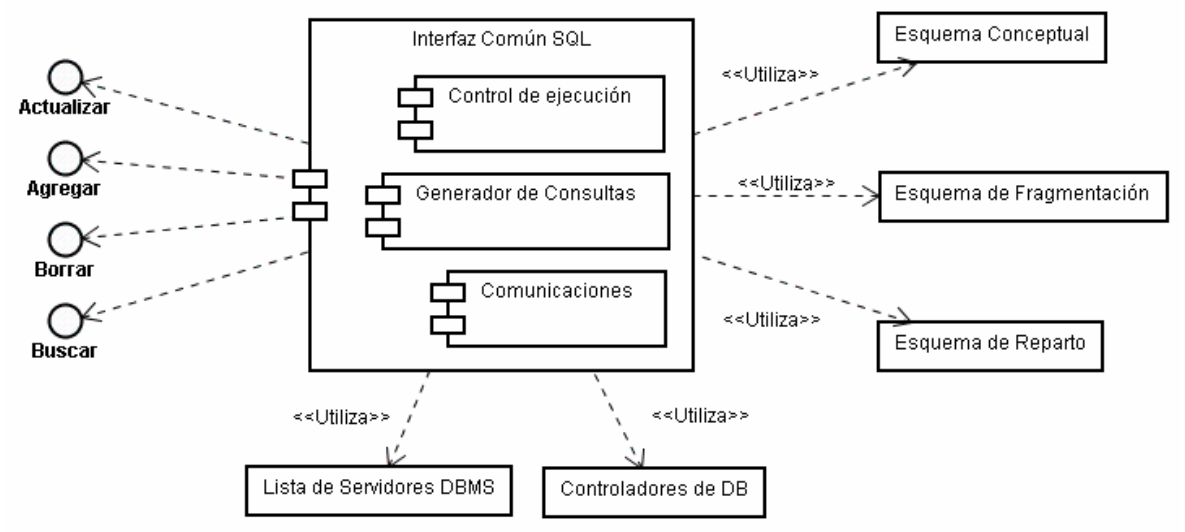

Figura 3.7. Estructura de la Interfaz Común SQL y módulos principales.

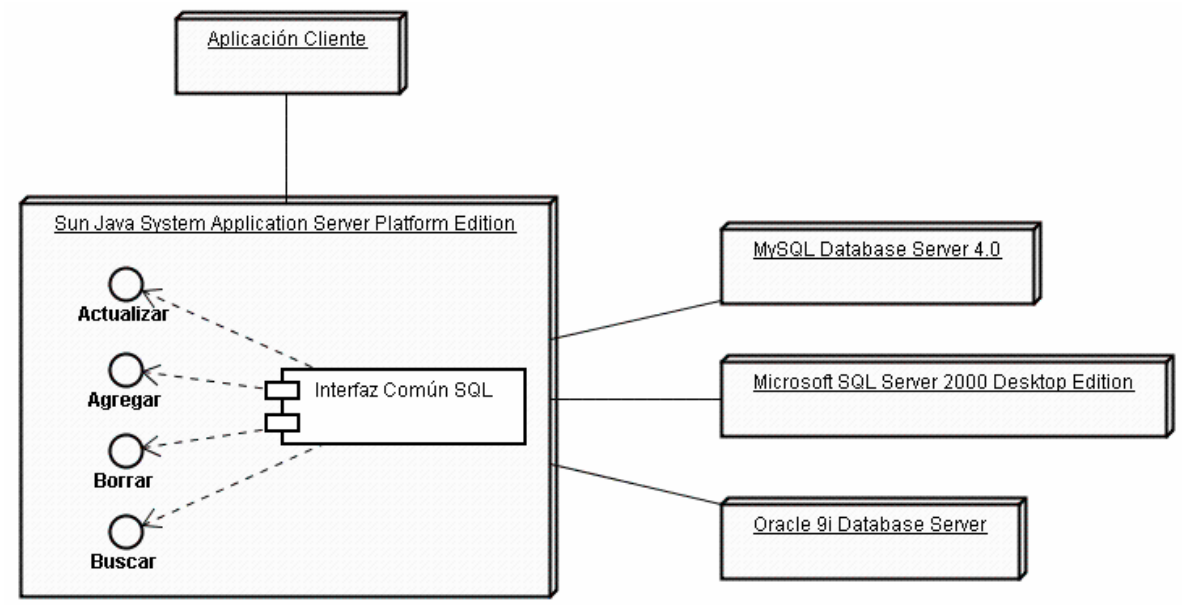

Figura 3.8. Despliegue de la Interfaz Común SQL.

Con el fin de que la "Interfaz Común SQL" pueda atender varias peticiones realizadas por los usuarios finales a través de una aplicación Cliente, necesita estar dentro de la capa intermedia de un modelo arquitectónico de sistemas distribuidos de 3 capas. (Vea la Figura anterior 3.8). En el campo de acción experimental este nivel se encuentra

representado por el servidor de aplicaciones y de páginas Web. De esta manera, este componente queda como el punto de unión entre el usuario final que utiliza la Aplicación Cliente y los datos almacenados dentro de los motores relacionales seleccionados en la red de computadoras.

El procedimiento general que la “Interfaz Común SQL” ejecuta al momento de ser utilizada por la aplicación Cliente se muestra en la siguiente Figura:

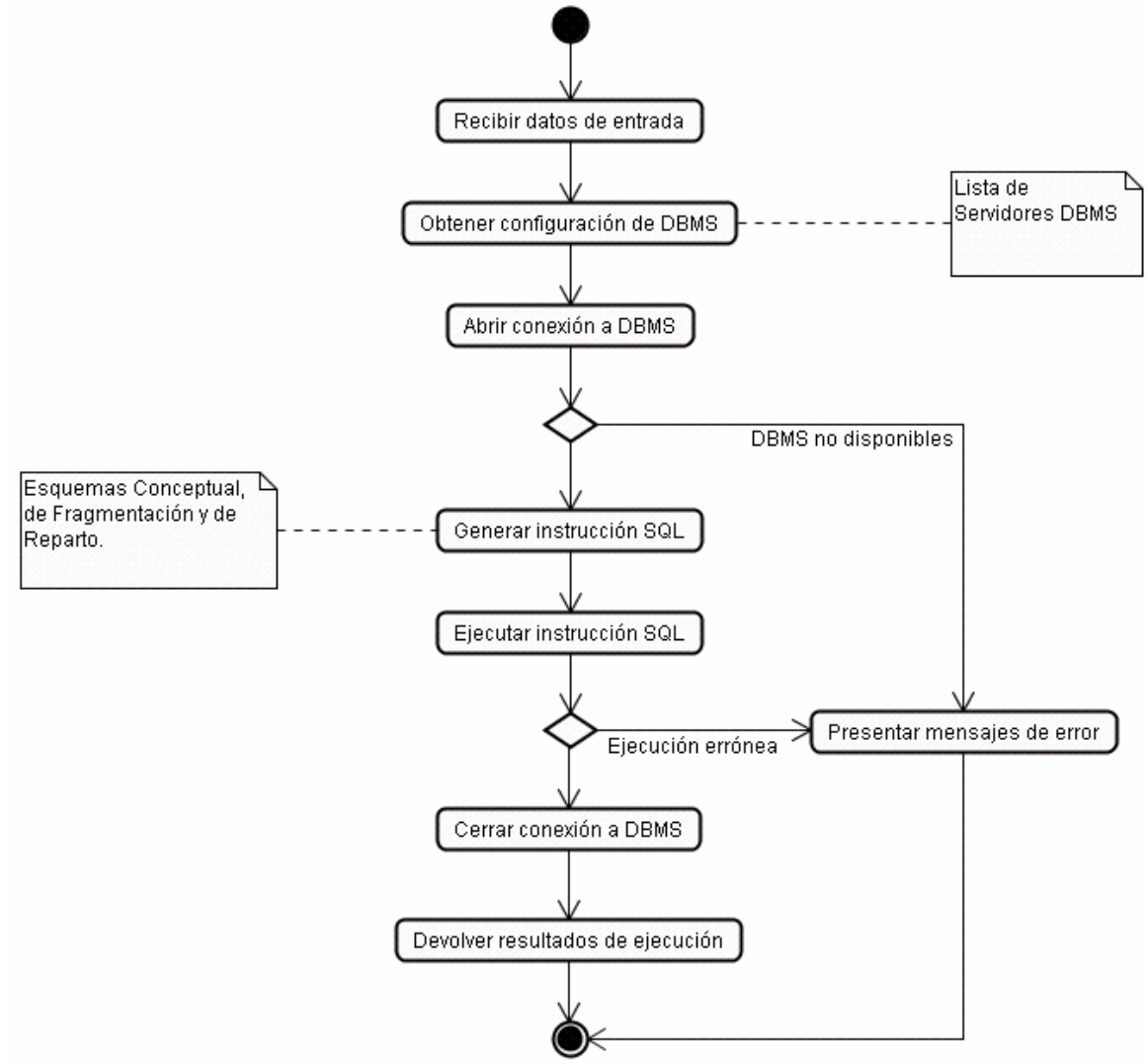

Figura 3.9. Actividades generales de la Interfaz Común SQL.

Para poder realizar sus actividades generales, la “Interfaz Común SQL” pone a disposición de las aplicaciones Cliente métodos de acceso público para que sean invocados. Éstos son llamados, desde el punto de vista OO, interfaces. Cada una de ellas tiene sus parámetros de entrada y son los que se utilizan a lo largo de todos los procedimientos internos para obtener la configuración de los Servidores de la base de datos distribuida, la generación y ejecución de las instrucciones SQL y, la devolución de los resultados o mensajes de error, si aplica.

Los métodos de acceso público a los que se acceden pertenecen al módulo “Control de ejecución” y éste, a su vez, determina la acción que se realiza a continuación dependiendo de la elección del usuario final acerca de que desea hacer con la información almacenada dentro de la base de datos distribuida. En el siguiente diagrama (Figura 3.10) se aprecia la clase “Control” y el resto de los módulos que colaboran con él.

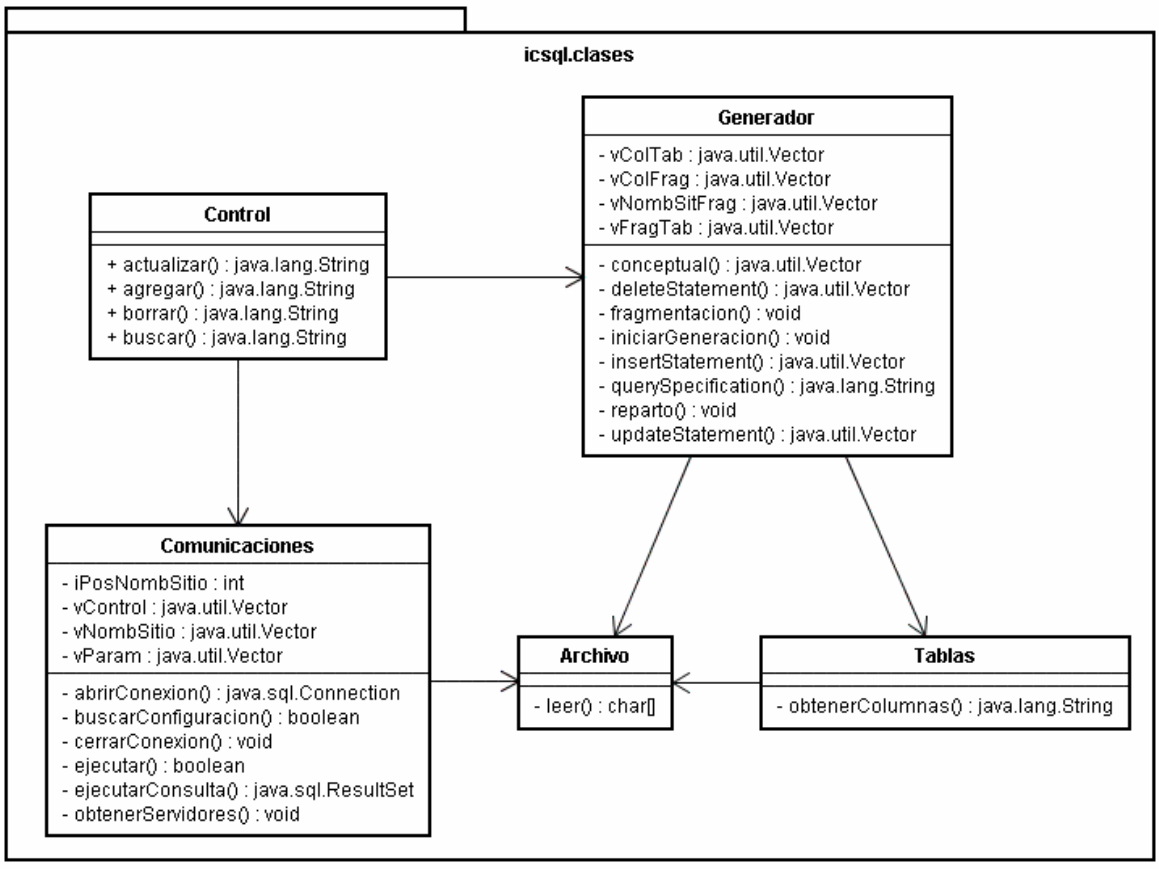

Figura 3.10. Vista estática de la Interfaz Común SQL.

La ejecución de las actividades generales dentro de la Interfaz Común SQL da forma al comportamiento dinámico de este *middleware*. Para tener una mayor referencia acerca de procedimientos realizados, el código fuente de este componente se encuentra en la sección A.1 del Apéndice

III.3. Comportamiento Dinámico.

La “Interfaz Común SQL” interacciona con la interfaz de usuario por medio de la técnica de ejecución de métodos remotos RPC solicitados desde la aplicación Cliente. Este procedimiento utiliza a los módulos o clases que integran al *middleware* para obtener las instrucciones SQL necesarias y que junto con los datos de entrada dan forma a ciertas sentencias de obtención y/o manipulación de datos que serán enviadas a los motores relacionales de los Servidores involucrados en éstas para que sean ejecutadas. Una vez que los Servidores de la base de datos distribuida terminen de ejecutar las instrucciones enviadas por la “Interfaz Común SQL” entonces se retoma el procedimiento anterior de manera inversa. Esto quiere decir, que cada motor relacional envía al *middleware* los resultados de la ejecución de cada instrucción procesada dentro de ellos y a través de “Control de ejecución” se retornan a la interfaz de usuario notificándole el acierto o error de sus transacciones.

Por otra parte, el equilibrio entre los procesos de las aplicaciones distribuidas fue obtenido haciendo que el componente Cliente se encargue de la traducción y arreglo de los datos recibidos desde el *middleware* para dar una apariencia uniforme a la información. De esta manera, la “Interfaz Común SQL” se encuentra preparada para conectarse con otros motores relacionales, aparte de los tomados en cuenta en esta investigación, asegurando sus futuras expansiones y alcances. Además, está el hecho de que emplea versiones estándares de las instrucciones seleccionadas de SQL-92 anteriormente definidas. Lo único que tendría que hacerse para utilizar este componente en otra aplicación distribuida es que la nueva aplicación Cliente solicite la ejecución de los métodos públicos de la clase “Control”, como se muestra en la Figura 3.10, plasmar los nuevos esquemas conceptual, de fragmentación y de reparto acerca de la base de datos distribuida a utilizar, agregar los nuevos parámetros de

conexión a los motores relacionales y, realizar los procedimientos necesarios para desplegar la información en la pantalla, lo cual es concerniente al diseño de una interfaz de usuario.

El estado general de la “Interfaz Común SQL” al ejecutar sus procesos se comporta de la siguiente manera:

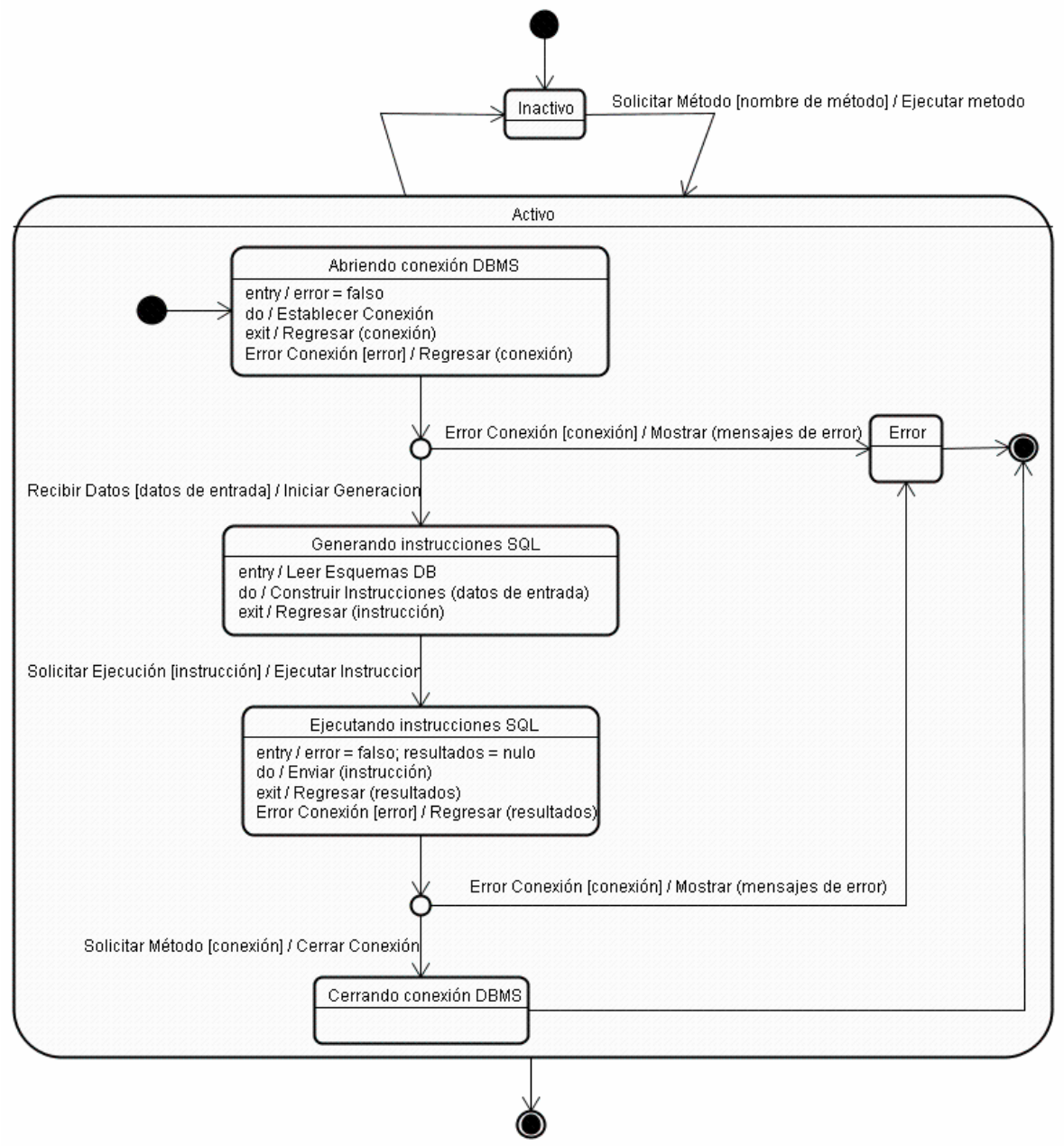

Figura 3.11. Estado general de la Interfaz Común SQL durante su ejecución.

La Figura 3.11 muestra el estado general de este componente al momento de ser ejecutado, el estado de actividad que tiene durante ese tiempo y su vuelta a la inactividad, esperando ser utilizado de nueva cuenta por otro proceso de las aplicaciones Cliente.

III.3.1. Módulo Control de ejecución.

Este es el módulo principal de la “Interfaz Común SQL”. Contiene los métodos de acceso público que las aplicaciones Cliente utilizan como primer paso a la obtención y/o manipulación de los datos almacenados en la base de datos distribuida. Cada uno de los métodos que la forman interactúa con las demás clases que conforman al *middleware*. Su funcionalidad se puede observar en la Figura 3.6.

Las actividades de cada uno de los casos de uso son muy similares a las actividades generales de la “Interfaz Común SQL”. Estas son:

- Recibir los datos de entrada que se utilizarán en las instrucciones SQL que se generen.
- Recuperar de los archivos de configuración la Lista de los Servidores DBMS que conforman a la base de datos distribuida.
- Obtener, verificar y abrir una conexión con los motores de base de datos para que, en el caso de algún error en ellos o simplemente no se encuentren funcionando o activos, indique al usuario final esa condición.
- Generar la instrucción SQL necesaria para el proceso utilizando los datos de entrada tanto de búsqueda de la información a actualizar como los nuevos datos a integrar en ella. Para este paso se requiere la participación del módulo Generador.
- Ejecutar la instrucción SQL. Para este paso se requiere la participación del módulo Comunicaciones para realizar esta acción y esperar los resultados de su ejecución.
- Cerrar la conexión. Paso necesario para que la comunicación se dé por concluida después de realizar la actualización de los datos.

- Devolver los resultados de la ejecución y/o mostrar mensajes de error, si aplica.

III.3.1.1. Caso de uso Actualizar.

Esta funcionalidad permite que la "Interfaz Común SQL" realice el proceso de actualización de datos sobre una o más tablas que componen a la base de datos. Los datos de entrada provienen de la aplicación Cliente. La secuencia de estas actividades y su colaboración con el resto de las clases se muestran en la Figura 3.12.

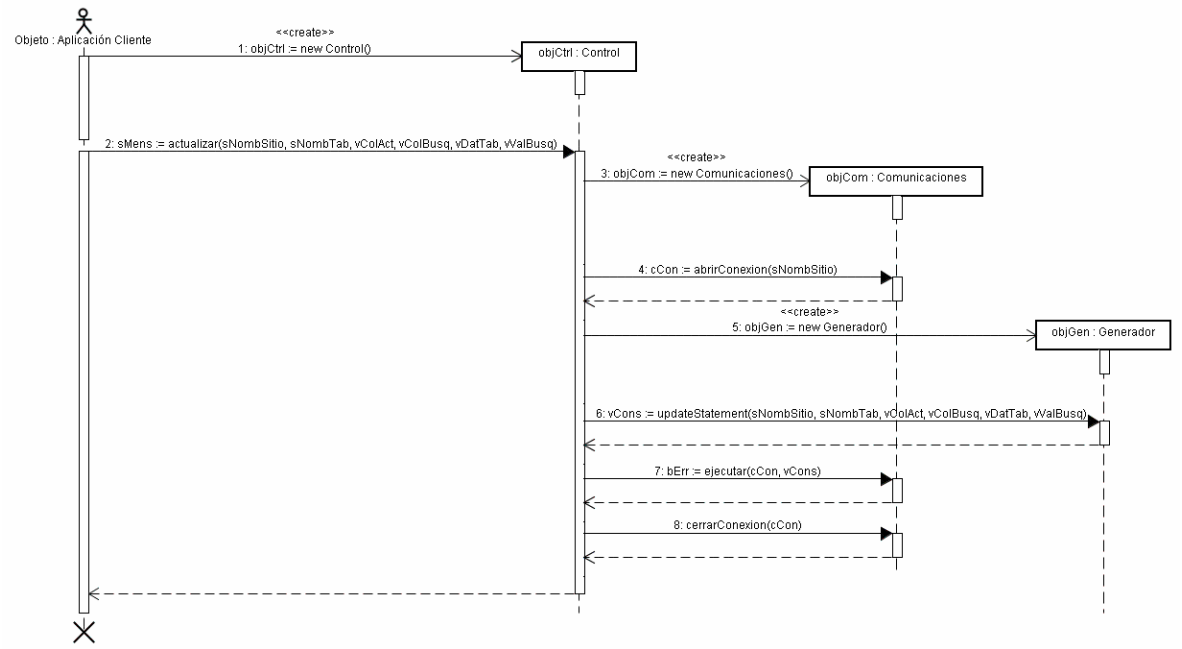

Figura 3.12. Diagrama de secuencia para Actualizar.

III.3.1.2. Caso de uso Agregar.

Este caso sirve para que la "Interfaz Común SQL" reciba los datos de entrada desde una aplicación Cliente y realice la inserción de esos nuevos datos dentro de una base de datos distribuida. Al igual que el caso de uso anterior, la secuencia de sus actividades y la colaboración que obtiene del resto de las clases se muestran en la siguiente figura:

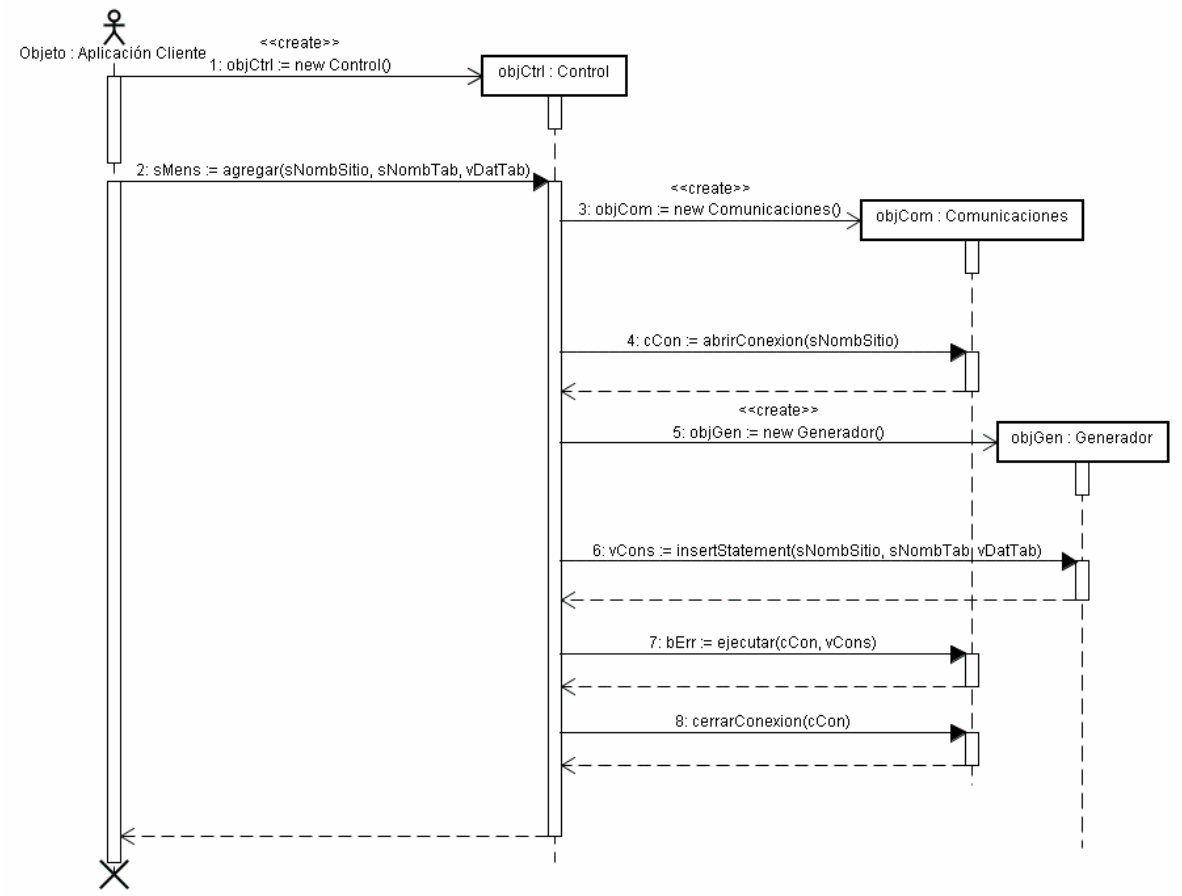

Figura 3.13. Diagrama de secuencia para Agregar.

III.3.1.3. Caso de uso Borrar.

Con este caso de uso, la "Interfaz Común SQL" permite a las aplicaciones Cliente, el poder borrar información de la base de datos distribuida ubicándola por medio de ciertos datos de entrada. La secuencia de operaciones para hacer esto posible se muestra en la Figura 3.14.

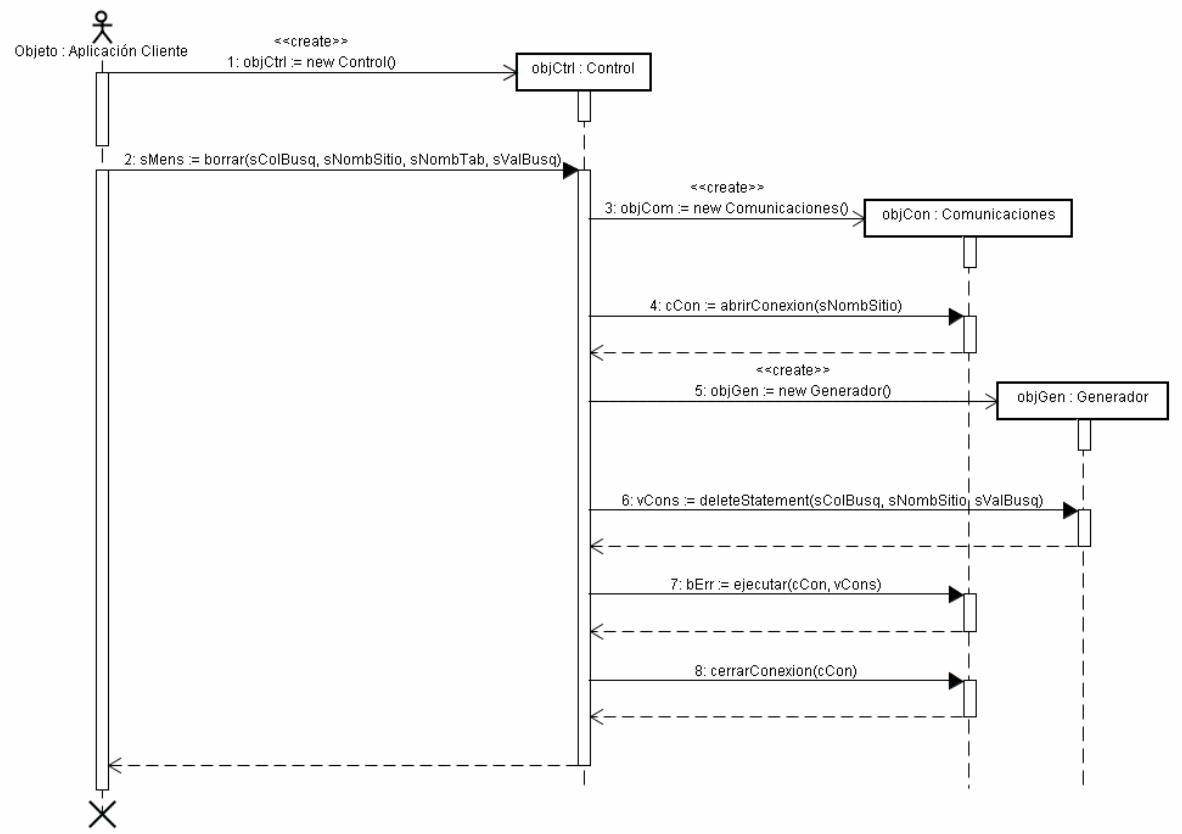

Figura 3.14. Diagrama de secuencia para Borrar.

III.3.1.4. Caso de uso Buscar.

El caso de uso Buscar difiere muy poco en sus actividades principales a los anteriores casos de uso pertenecientes al módulo “Control”. La razón: es posible que una búsqueda de información requiera la participación de todos los servidores de bases de datos relacionados por medio del esquema conceptual único y, aunque en los demás casos de uso principales de la “Interfaz Común SQL” se puedan ejecutar varias instrucciones con esta condición, aquí es importante que, si un servidor que posee cierta información de interés para el usuario no se encuentra disponible, la ejecución del resto de las sentencias de obtención de datos no debe de detenerse.

La fragmentación mixta de algunas tablas hace que por una instrucción de obtención de información se requieran como mínimo más de una, derivadas de esta última, que realice la misma operación solo dirigida a diferente fragmento y/o servidor en donde se

encuentre. La Figura 3.15 muestra el diagrama de secuencia de este caso de uso y se puede apreciar que por cada vez que la aplicación Cliente necesite obtener datos, el módulo "Generador" hará más de una instrucción de búsqueda, tantas como fragmentos den forma a la tabla del esquema conceptual.

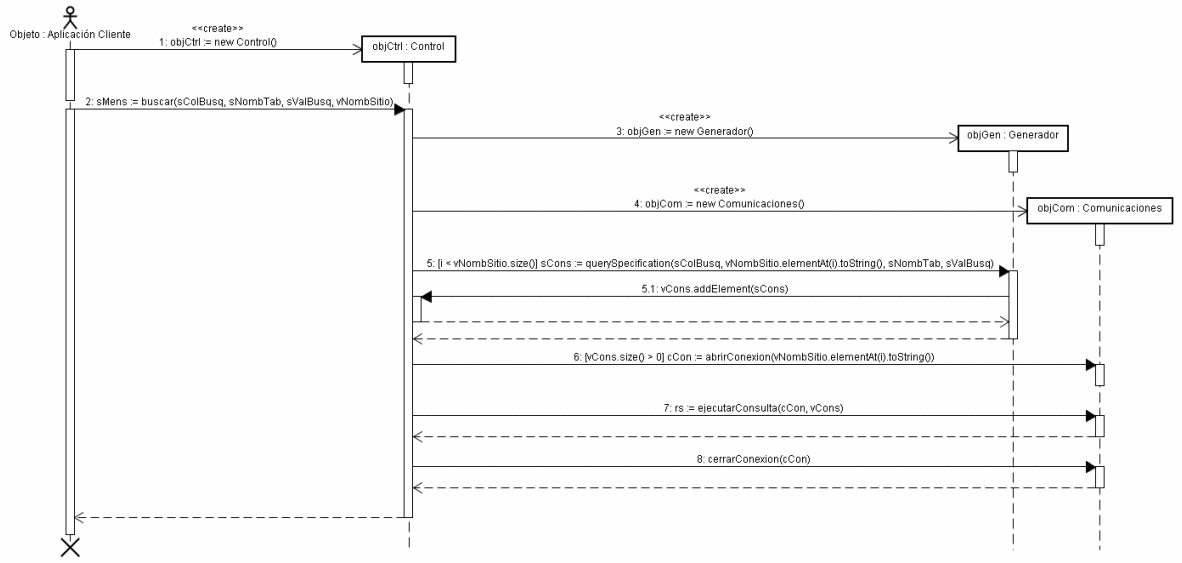

Figura 3.15. Diagrama de secuencia para Buscar.

III.3.2. Módulo Archivo.

Este elemento es utilizado por los módulos "Generador de consultas" y "Comunicaciones" para realizar sus actividades y proporcionar sus servicios a la "Interfaz Común SQL". El objetivo de esta clase es la de permitir la lectura de los archivos de configuración que se le proporcionen, es decir, los Esquemas Conceptual, de Fragmentación y de Reparto de la base de datos distribuida. Ver Figura 3.16.

Al obtener el nombre del archivo de configuración a leer, este módulo prepara una variable que contendrá el contenido del mismo y la prepara para que sea devuelta al método que requiere de esta acción. Las actividades que realiza este módulo se pueden ver en la Figura 3.17.

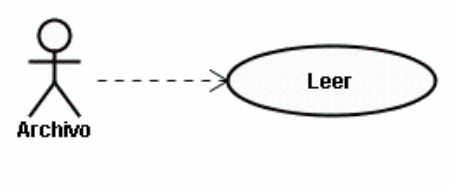

Figura 3.16. Funcionalidad de la clase Archivo.

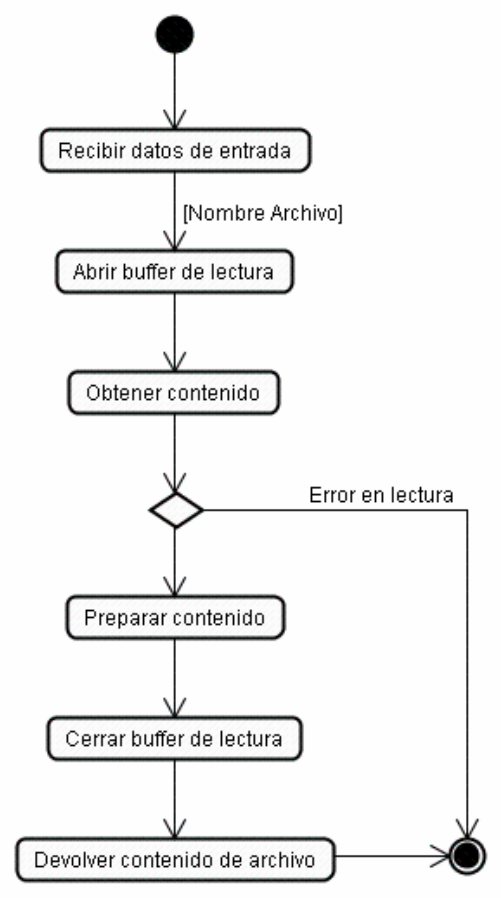

Figura 3.17. Actividades de la clase Archivo.

III.3.3. Módulo Tablas.

Este módulo es utilizado únicamente por el módulo “Generador de consultas”. Sirve como elemento auxiliar al momento de acceder al Esquema de Fragmentación de la base de datos distribuida y únicamente obtiene las columnas de cada uno de los fragmentos

que componen a una tabla en el Esquema Conceptual. Requiere de la participación de la clase Archivo descrita anteriormente. Ver Figura 3.18.

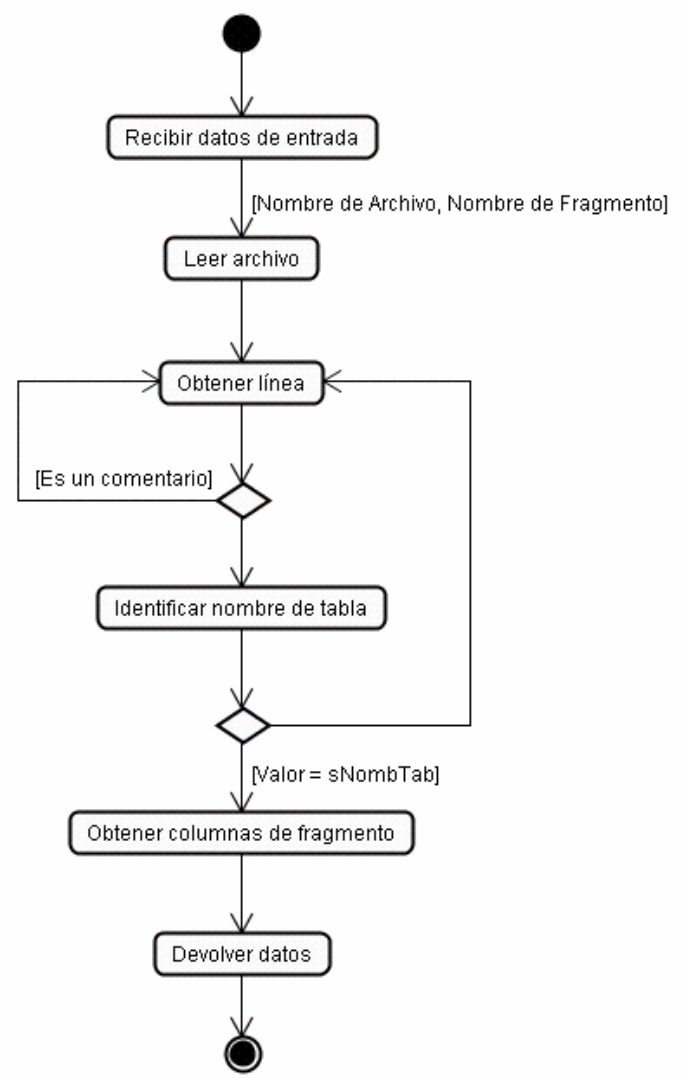

Figura 3.18. Actividades de la clase Tablas.

III.3.4. Módulo Generador de consultas.

En el campo de acción experimental se estableció el uso de tres distintos DBMS relacionales: Microsoft SQL Server, Oracle Database Server y MySQL Database Server. Cada uno de estos sistemas tiene una cierta compatibilidad con SQL. El primero se basa en

SQL-92 y los otros dos en SQL-99. La utilización del mismo lenguaje normalmente aseguraría que la ejecución de las instrucciones arrojarían los mismos resultados y, sin embargo, no es así. Por ejemplo, la sentencia “SELECT” del motor relacional Oracle difiere en gran medida de la que implementa el DBMS MySQL. Aunque utilizan la misma versión de SQL, las funcionalidades extras añadidas al mismo por parte de estas empresas (llamadas “extensiones SQL”) hace que ni entre sí puedan intercambiar información fácilmente, ni que decir cuando las versiones de SQL son distintas en el caso de utilizar a Microsoft SQL Server. Lo más recomendable en estos casos es la utilización de la gramática estándar de SQL definida por ANSI/ISO.

Pero, ¿qué versión del estándar? SQL-92 es el conglomerado del Lenguaje de Base de Datos desde su primera versión. Representa una madurez en la aplicación de los conceptos relacionales y presenta una gran cantidad de tipos de datos que ayudan a representar con mucha fidelidad los requerimientos de cualquier problema a resolver utilizando una base de datos. Por su parte, SQL-99 es una evolución de estos conceptos pero a un nivel Orientado a Objetos porque existen nuevos tipos de datos acoplables con esta tecnología de desarrollo de aplicaciones y, aunque la tendencia es esta ideología, la utilización de dos distintas versiones de un mismo lenguaje dentro de una base de datos distribuida trae consigo muchas incompatibilidades entre sí. Como la versión de 1999 respeta mucho de la estructura de la versión de 1992, pues la elección de esta última como plataforma para la obtención y/o manipulación de datos no resta efectividad a la funcionalidad de la “Interfaz Común SQL”.

Por motivos de investigación y de la extensa gramática del Lenguaje para Base de Datos SQL-92, este módulo genera versiones reducidas de las variantes elegidas de las instrucciones de obtención y/o manipulación de datos que más utilizan y, que con ayuda de la interfaz de usuario, se logran los resultados deseados.

El módulo “Generador de consultas” es utilizado por la “Interfaz Común SQL” para la generación de instrucciones de SQL-92 estándar que serán ejecutadas en los distintos servidores de base de datos. Su funcionalidad se representa en la Figura 3.19.

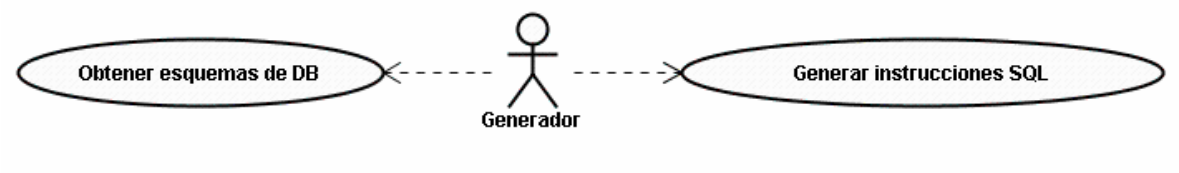

Figura 3.19. Funcionalidad de la clase Generador.

Entre sus actividades se encuentran:

- Leer el Esquema Conceptual, de Fragmentación y de Reparto de la base de datos distribuida para la correcta determinación y ubicación de estos elementos a lo largo de la infraestructura del campo de acción experimental. Por diseño, la lectura de los tres esquemas es imperativa para cualquier clase de sentencia requerida.
- Generar una o más instrucciones que obtengan y/o manipulen información en la base de datos dependiendo de su estructura plasmada en los esquemas anteriores.

III.3.4.1. Caso de uso Obtener esquemas DB.

La finalidad de tener los esquemas Conceptual, de Fragmentación y de Reparto en archivos de configuración externos a la “Interfaz Común SQL” permite la estandarización de la forma de acceso a esta información aunque es posible obtenerla sin necesidad de estos elementos. A través de rutinas de programación, los DBMS pueden proporcionarla, sin embargo, la utilización de tres motores relacionales de base de datos distintos entre sí hace que el formato y la estructura de la misma sean diferentes. La interpretación de esta información podría llevar a errores en el procedimiento lo que podría provocar un fallo general en la aplicación distribuida, la obtención y/o manipulación errónea de datos que afectarían a los requerimientos del usuario final, que generarían errores dentro de los sistemas de bases de datos, etcétera.

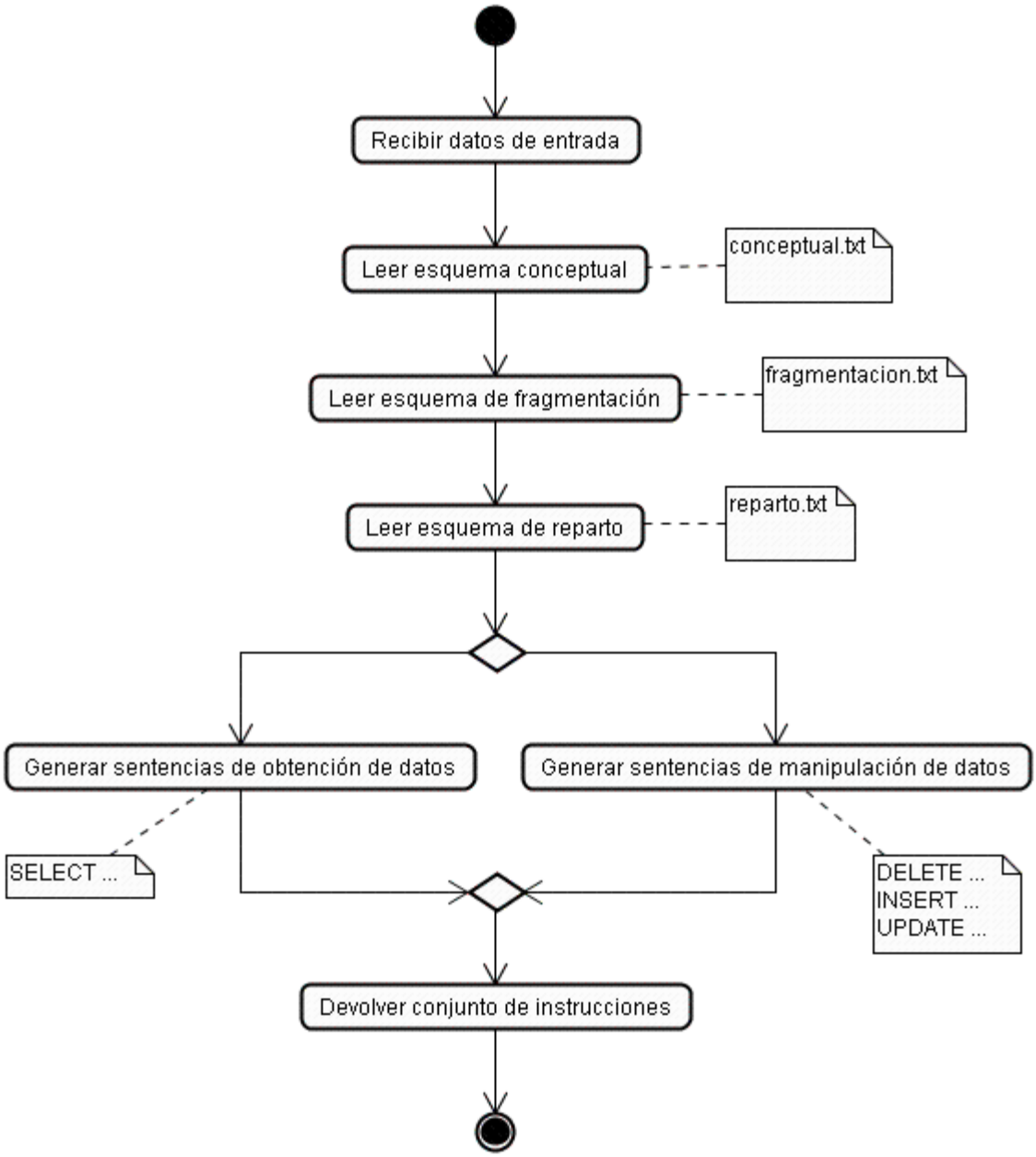

Figura 3.20. Actividades de la clase Generador.

El Esquema Conceptual, representado en la Figura 3.20 como "conceptual.txt" es un archivo de texto con un formato legible por los OS Microsoft Windows y Linux por igual a través del módulo "Archivo". Representa la forma en la que está construida la base de datos y la manera en la cual el usuario final a través de la interfaz de usuario la percibe. Cada línea de configuración puede tener la siguiente estructura:

```
[[REM [<Comentario>]]
     <Tabla> = (<Columna> [, <Columna>])]
```

- El valor `REM` indica que por cada línea de texto puede existir un `<Comentario>` que ayuda a una mejor comprensión del contenido del archivo en general.
- El campo `<Tabla>` se refiere al nombre conceptual de la tabla en la base de datos o la manera de nombrarla en el esquema lógico de una base de datos cualquiera. Sirve de referencia puesto que de ella se derivan los nombres de los fragmentos que la componen.
- El campo `<Columna>` indica el nombre de cada columna que compone a la tabla en el esquema lógico de la base de datos. Es importante la escritura de todas las columnas separadas por una coma y encerradas entre paréntesis.

El Esquema de Fragmentación posee la configuración sobre cuáles son los fragmentos de cada tabla conceptual y los campos que posee cada uno de ellos. Se representa en la Figura 3.20 como “fragmentación.txt” y, al igual que los demás archivos de configuración se lee por medio del módulo “Archivo”. Posee la siguiente estructura:

```
[[REM [<Comentario>]]
     <Tabla> = <Fragmento> [, <Fragmento>]]
[[REM [<Comentario>]]
     <Fragmento> = (<Columna> [, <Columna>])]
```

- En este archivo de configuración, el campo `<Fragmento>` indica el nombre de cada uno de los segmentos que componen a la tabla del esquema conceptual de la que se deriva. Los nombres de cada uno de ellos deben estar separados por una coma.

También se requiere del archivo “reparto.txt”; vea la Figura 3.20. Éste contiene la representación del Esquema de Reparto y ubica en dónde, es decir, en qué servidor que compone a la base de datos distribuida se encuentran los fragmentos de las tablas identificadas en el esquema conceptual. Su formato es el siguiente:

```
[[REM [<Comentario>]]
     <Sitio> = <Fragmento> [, <Fragmento>]]
```

- El campo `<Sitio>` indica un nombre representativo del servidor de base de datos en la red para que la “Interfaz Común SQL” lo ubique y seleccione el controlador de base de datos necesario para conectarse a él.

III.3.4.2. Caso de uso Generar instrucciones SQL.

Los casos de uso que se muestran en la Figura 3.6 representan las acciones que la “Interfaz Común SQL” realiza, a petición de una aplicación Cliente, sobre los datos almacenados dentro de los motores relacionales elegidos. Estas acciones se dividen en dos grupos:

- Obtención de datos. El grupo a la que pertenece la instrucción SQL denominada “SELECT”.
- Manipulación de datos. El grupo de instrucciones SQL a la que pertenecen “DELETE”, “INSERT” y “UPDATE”.

La generación de éstas se realizan dentro de la clase “Generador” a través de los métodos deleteStatement(), insertStatement(), querySpecification() y updateStatement(), mostrados en la Figura 3.10. Cada uno de ellos genera una o más instrucciones dependiendo de los datos de entrada y de la estructura de la base de datos distribuida.

La estructura de las instrucciones SQL-92 elegidas se muestra en el Cuadro 3.1, sin embargo, cabe aclarar que se utiliza la segunda variante de las instrucciones “DELETE”

y “UPDATE”. Esto significa que son instrucciones que actúan sobre datos “posicionados” o que requieren de una condición de búsqueda para realizar operaciones sobre ellos.

Cuadro 3.1. Estructura original de instrucciones SQL-92.

ESTRUCTURA DE INSTRUCCIÓN	COMENTARIO
DELETE FROM *Nombre de la tabla* [WHERE *Condición de búsqueda*]	Elimina todas las filas de cierta tabla que cumplan una condición de búsqueda proporcionada por el usuario.
INSERT INTO *Nombre de la tabla Columnas y fuente de inserción*	Esta sentencia agrega a la tabla señalada nuevos valores para cada columna especificada y proveniente de una fuente de origen de datos.
SELECT [*Cuantificador*] *Lista de selección Expresión de tabla*	Esta instrucción busca a lo largo de la estructura de datos de origen elegida los valores que contienen las filas de algunas o todas las columnas indicadas que las componen y los selecciona dependiendo de la condición de búsqueda utilizada para este fin.
UDPATE *Nombre de la tabla* SET *Lista de asignación de valores a columnas* [WHERE *Condición de búsqueda*]	Lo cual quiere decir que esta sentencia busca a lo largo de la tabla elegida las filas que cumplan la condición de búsqueda proporcionada y cambia los valores anteriores de las columnas señaladas por unos nuevos.

Dentro de las operaciones de manipulación de datos, la sintaxis de la instrucción “DELETE” que se genera en este módulo es la siguiente:

```
DELETE FROM <Nombre de Tabla>
WHERE <Expresión de búsqueda>
```

- `<Nombre de Tabla>`, indica el nombre de la tabla a que se utiliza en la instrucción SQL que se genere. Puede ser también el nombre de un fragmento.
- `WHERE`. Normalmente, el lenguaje de base de datos SQL-92 determina que esta cláusula puede o no ser utilizada, sin embargo, las aplicaciones distribuidas de base de datos requieren tomar datos de uno o más fragmentos, por lo qué el uso de esta sentencia es primordial, es decir, no es opcional.
- `<Expresión de búsqueda>`, contiene una expresión que se necesita para ubicar cierta información que se solicite. Se compone internamente de `“<Nombre de Tabla>.<Nombre de Columna> = <Valor de búsqueda>”`. En las instrucciones de base de datos distribuida, al menos debe de existir una expresión de búsqueda.

- `<Nombre de Columna>`, es el nombre de la columna que pertenece a una tabla o fragmento de la base de datos distribuida.
- `<Valor de búsqueda>`, es el valor de la Columna que tiene que coincidir con los datos seleccionados y devueltos a la aplicación Cliente desde el motor de base de datos.

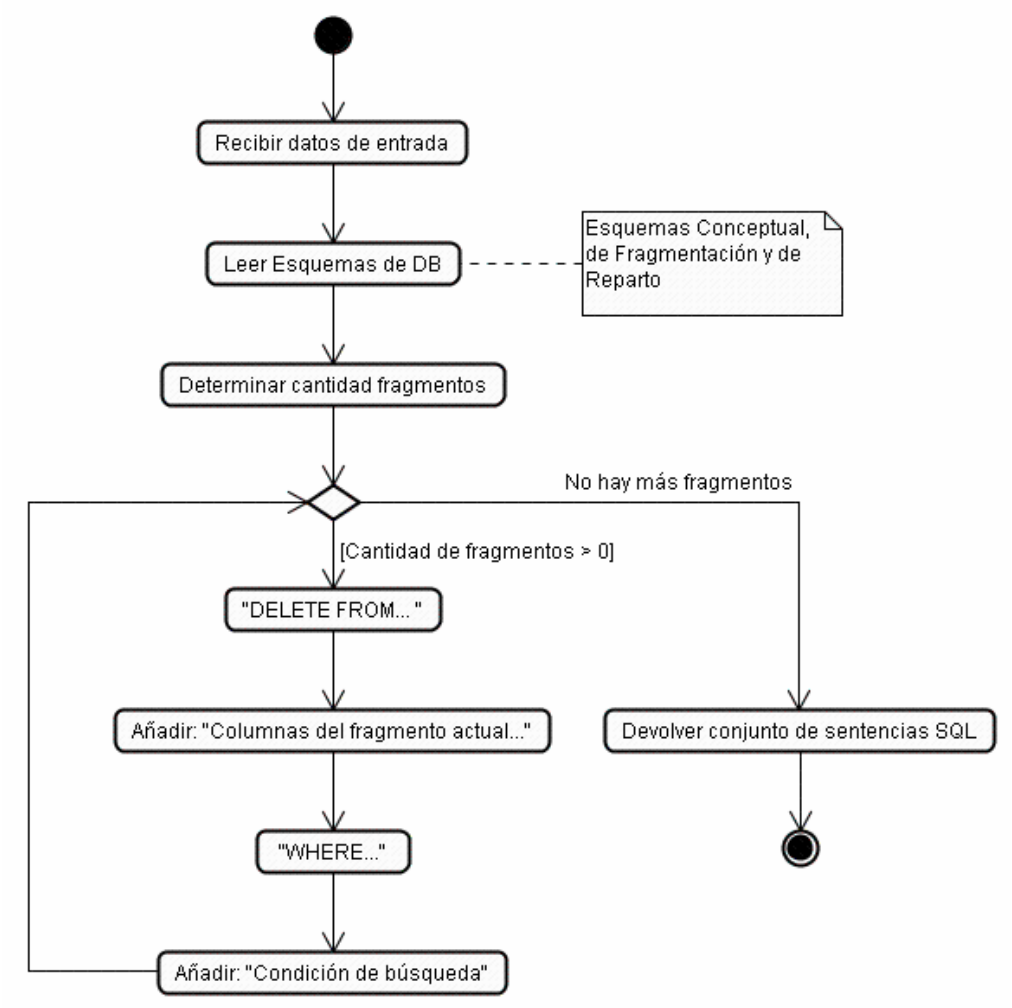

Figura 3.21. Actividades para generar la instrucción DELETE.

Normalmente, las instrucciones de los ambientes distribuidos pueden citar más de una tabla o fragmento, especialmente para la fragmentación vertical, generando otra instrucción con el siguiente nombre de tabla o fragmento involucrado.

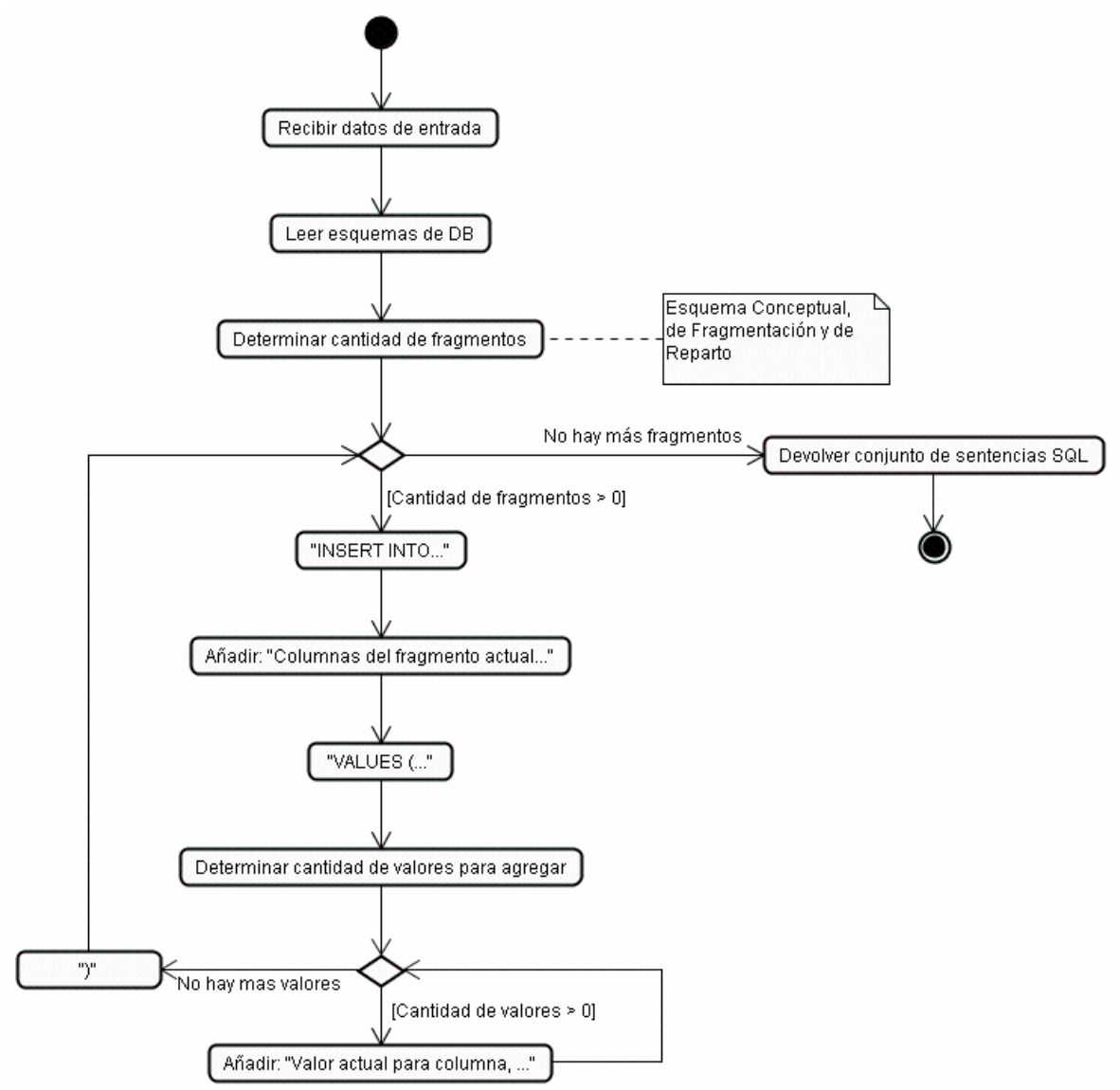

Figura 3.22. Actividades para generar la instrucción INSERT.

Otra instrucción de manipulación de los datos dentro de una base de datos distribuida es la instrucción “INSERT”. La Figura anterior (3.22), muestra las actividades necesarias para generar esta instrucción.

La sintaxis que este módulo utiliza para generarla es la siguiente:

```
INSERT INTO <Nombre de Tabla>
(<Nombre de Columna> [, <Nombre de Columna>,…])
```

```
VALUES (<Valor a agregar> [, <Valor a agregar>…])
```

- <Nombre de Tabla>, indica el nombre de la tabla a que se utiliza en la instrucción SQL que se genere. Puede ser también el nombre de un fragmento.
- <Nombre de Columna>, es el nombre de la columna que pertenece a una tabla o fragmento de la base de datos distribuida.
- <Valor a agregar>, es el nuevo valor a ser integrado a la tabla de la base de datos distribuida. cada valor tiene que corresponder al tipo de dato almacenado y debe existir uno por cada columna de la tabla, aunque el valor a guardar sea nulo o NULL.

La sintaxis utilizada por la “Interfaz Común SQL” de la instrucción “SELECT” es la siguiente:

```
SELECT *
FROM <Nombre de Tabla> [, <Nombre de Tabla>]
WHERE <Expresión de búsqueda> [AND <Expresión de búsqueda>]
```

- El símbolo *, índica que obtendrá la información de todas las columnas de la tabla que se le indique. La “Interfaz Común SQL” internamente utiliza el formato “<Nombre de Tabla>.<Nombre de Columna>” para citar a cada una de ellas.
- <Nombre de Tabla>, indica el nombre de la tabla a que se utiliza en la instrucción SQL que se genere. Puede ser también el nombre de un fragmento.
- WHERE. Normalmente, el lenguaje de base de datos SQL-92 determina que esta cláusula puede o no ser utilizada, sin embargo, las aplicaciones distribuidas de base de datos requieren tomar datos de uno o más fragmentos, por lo qué el uso de esta sentencia es primordial, es decir, no es opcional.
- <Expresión de búsqueda>, contiene una expresión que se necesita para ubicar cierta información que se solicite. Se compone internamente de “<Nombre de Tabla>.<Nombre de Columna> = <Valor de

búsqueda>”. En las instrucciones de base de datos distribuida, al menos debe de existir una expresión de búsqueda. De manera opcional se puede integrar la palabra clave AND y una segunda expresión de búsqueda. Esto sirve para tener otra coincidencia entre los datos obtenidos por SQL.

- <Nombre de Columna>, es el nombre de la columna que pertenece a una tabla o fragmento de la base de datos distribuida.
- <Valor de búsqueda>, es el valor de la Columna que tiene que coincidir con los datos seleccionados y devueltos a la aplicación Cliente desde el motor de base de datos.

La Figura 3.23 muestra las actividades que se realizan para generar la instrucción “SELECT”.

Por último, la instrucción “UPDATE” es obtenida por el módulo “Generador de consultas” utilizando la siguiente sintaxis:

```
UPDATE <Nombre de tabla>
SET <Nombre de Columna a modificar> = <Nuevo valor>
     [, <Nombre de Columna a modificar> = <Nuevo valor>…]
WHERE <Expresión de búsqueda>
```

- <Nombre de Tabla>, indica el nombre de la tabla a que se utiliza en la instrucción SQL que se genere. Puede ser también el nombre de un fragmento.
- <Nombre de Columna a modificar>, hace referencia a la columna de la tabla a la que se le quieren modificar sus valores. Puede ser más de una. Es necesario que por cada una de ellas se le coloque su <Nuevo valor>.
- <Expresión de búsqueda>, contiene una expresión que se necesita para ubicar cierta información que se solicite. Se compone internamente de “<Nombre de Tabla>.<Nombre de Columna> = <Valor de búsqueda>”. En las instrucciones de base de datos distribuida, al menos debe de existir una expresión de búsqueda. De manera opcional se puede

integrar la palabra clave `AND` y una segunda expresión de búsqueda. Esto sirve para tener otra coincidencia entre los datos obtenidos por SQL.

- `<Nombre de Columna>`, es el nombre de la columna que pertenece a una tabla o fragmento de la base de datos distribuida.
- `<Valor de búsqueda>`, es el valor de la Columna que tiene que coincidir con los datos seleccionados y devueltos a la aplicación Cliente desde el motor de base de datos.

La Figura 3.24 muestra las actividades que el módulo “Generador de consultas” realiza para obtener la instrucción “UPDATE”.

III.3.5. Módulo Comunicaciones.

El módulo “Comunicaciones” tiene la responsabilidad de realizar todas las rutinas necesarias para abrir una conexión con cada uno de los Servidores de la base de datos distribuida, ejecutar las operaciones de obtención y/o manipulación de datos y cerrar la misma cada que se lo indique la “Interfaz Común SQL”.

Como se vio en la sección II.3.2. “El lenguaje de programación Java y bases relacionales de datos”, el uso de JDBC para la conexión con bases de datos permite que cualquier programa creado en este lenguaje pueda acceder a la información que almacenan las mismas por medio de un conjunto de procedimientos almacenados en un “Controlador JDBC” o *driver* dedicado, es decir, cada motor de base de datos requiere uno especialmente diseñado para él.

La “Interfaz Común SQL” a través del módulo “Comunicaciones” requiere la utilización de estos *drivers* para conectarse a los DBMS que le dan forma a la base de datos distribuida y ejecutar en ellos, las instrucciones de obtención y/o manipulación de datos que necesitan las aplicaciones Cliente.

Cada motor de base de datos para que admita conexiones remotas requiere de ciertos parámetros de configuración, definidos por Sun Microsystems, para asegurar las futuras extensiones o soporte a otros DBMS, además de los involucrados en el Campo de acción experimental, por ejemplo.

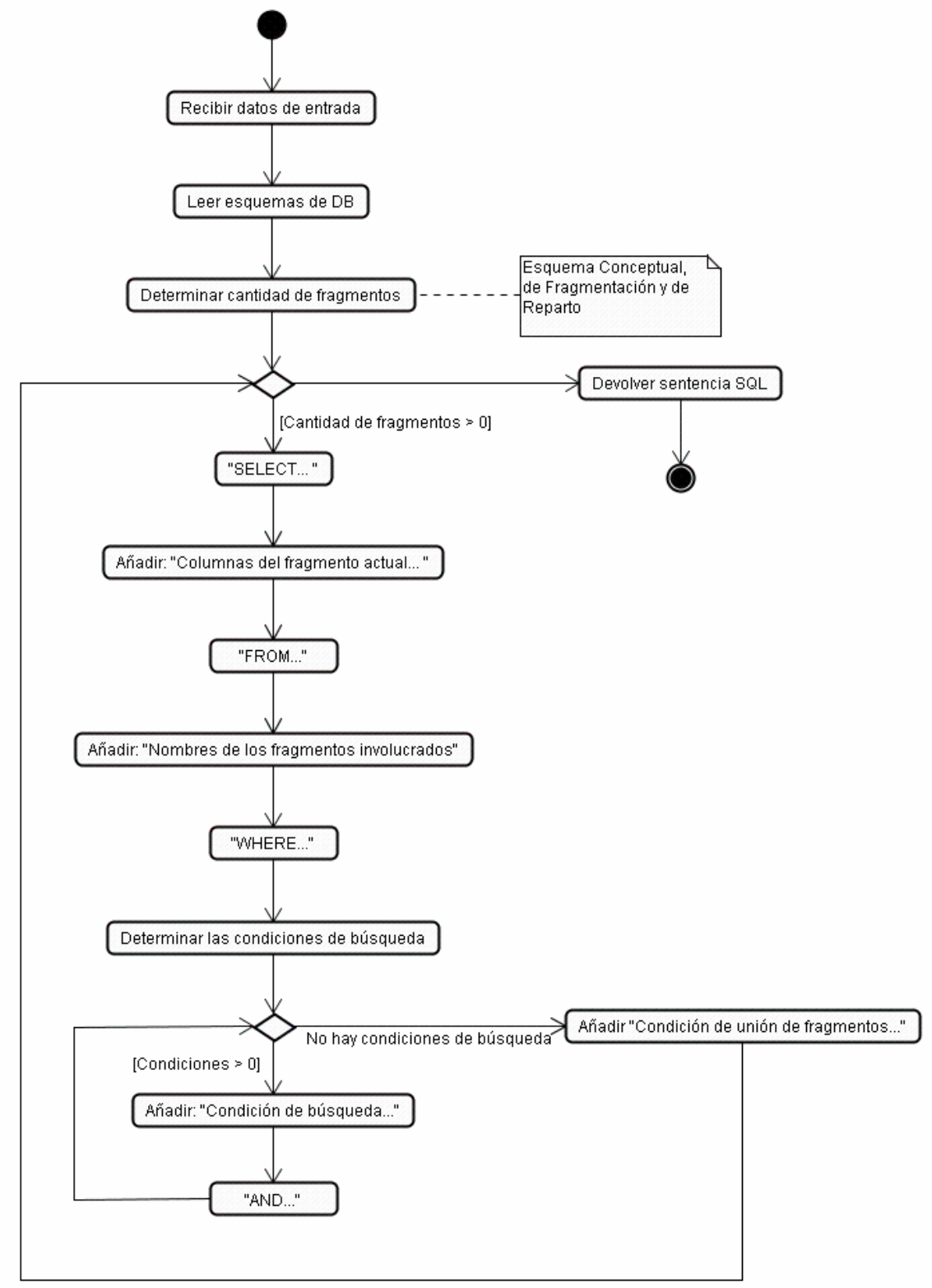

Figura 3.23. Actividades para generar la instrucción SELECT.

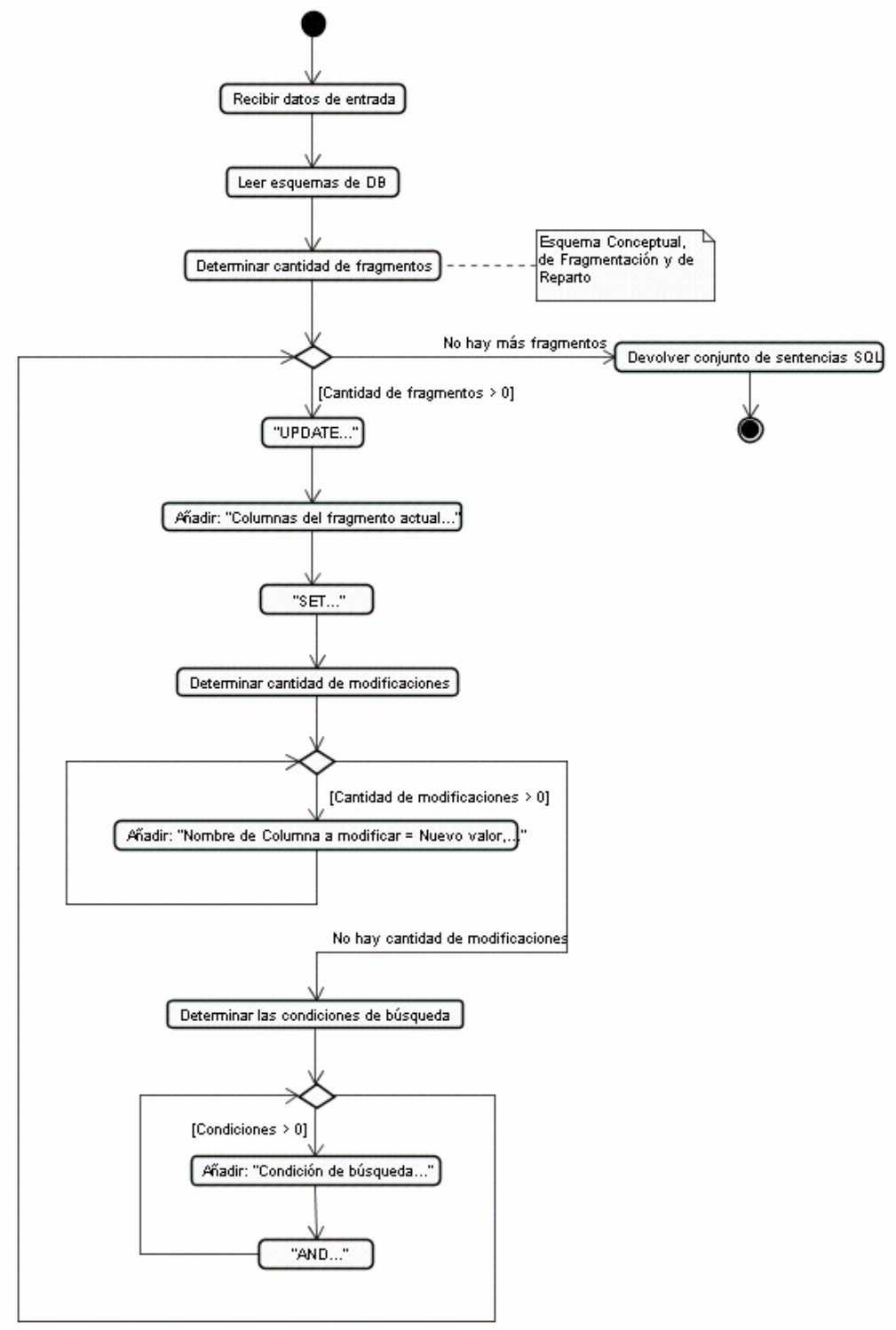

Figura 3.24. Actividades para generar la instrucción UPDATE.

La funcionalidad de la clase “Comunicaciones” se puede ver en la Figura 3.25 y en ella se resumen las actividades generales de la misma.

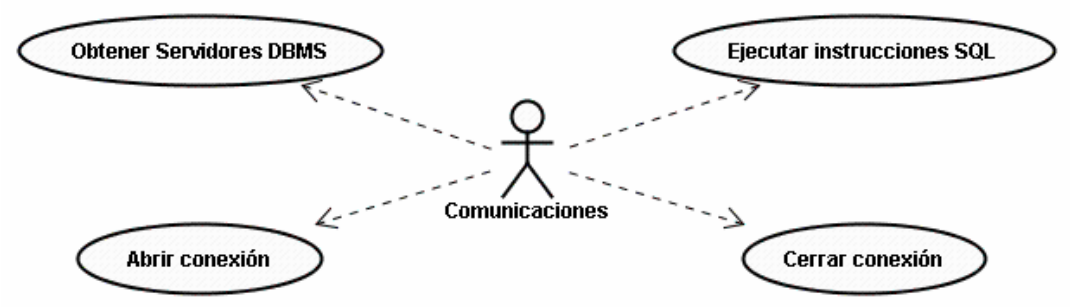

Figura 3.25. Funcionalidad de la clase Comunicaciones.

III.3.5.1. Caso de uso Obtener Servidores DBMS.

En esta sección de la clase “Comunicaciones” se realizan los procedimientos necesarios para obtener la Lista de los Servidores DBMS que soportan a la base de datos distribuida para que la “Interfaz Común SQL” pueda acceder y ejecutar en ellos las acciones que se requieran. Este elemento se representa en la Figura 3.7 como un elemento externo; realmente es un archivo de configuración conocido como “servidores.txt”.

Este archivo posee los parámetros de conexión necesarios para todo motor relacional que le de soporte a la base de datos del Esquema Conceptual. Su formato es el siguiente:

```
[[REM [<Comentario>]] <Sitio> *
     <Nombre de la clase Controlador> *
     <Cadena de Conexión>
```

- El valor `REM` indica que por cada línea de texto puede existir un `<Comentario>` que ayuda a una mejor comprensión del contenido del archivo en general.

- `<Sitio>` es el nombre representativo relacionado al Servidor DBMS que aloja ciertos fragmentos del Esquema Conceptual de una base de datos y que lo identifica unívocamente en la red de computadoras.
- `<Nombre de la clase Controlador>` es el nombre que le da el fabricante del motor de base de datos a la clase Java o *driver* que accede al DBMS.
- `<Cadena de Conexión>` es una cadena de texto que contiene los parámetros de conexión que requiere el motor de base de datos para permitir la obtención y/o manipulación de los datos que administra.

Sun Microsystems al diseñar a las API JDBC define que para conectarse a una base de datos desde Java se requiere el nombre del controlador que se utilizará para ello. El Administrador de Controladores de Java o *Driver Manager* necesita como parámetro el nombre de esa clase para iniciarle una instancia y realizar a través de ella, las operaciones de obtención y/o manipulación de datos. El Cuadro 3.2 indica cuáles son esos nombres de clase para los DBMS elegidos en el campo de acción experimental.

Cuadro 3.2. Nombres de clases "Controlador JDBC" para los DBMS elegidos.

NOMBRE DEL DBMS	NOMBRE DE CLASE "CONTROLADOR JDBC"
Microsoft SQL Server 2000 Desktop Edition	com.microsoft.jdbc.sqlserver.SQLServerDriver
MySQL Database Server 4.0.20	com.mysql.jdbc.Driver
Oracle 9i Database Server (9.0.1)	oracle.jdbc.OracleDriver

De la misma forma, Sun Microsystems determina que la Cadena de Conexión debe estar formada por los siguientes elementos:

- El Localizador Uniforme de Recursos de la Base de datos a utilizarse o *DataBase Uniform Resource Locator o URL*
- Nombre de usuario autorizado con privilegios de conexión y escritura de base de datos o *Username*.
- Contraseña de acceso del Usuario autorizado o *Password*.

Java define que el URL de acceso a una base de datos tiene la siguiente estructura, vea la Figura 3.26:

- El Protocolo de acceso. Por omisión éste se llama `jdbc`.
- El Subprotocolo. El nombre del controlador o el nombre del mecanismo de conectividad de la base de datos.
- El Subnombre. La manera de identificar la fuente de datos. Este subnombre puede variar dependiendo del fabricante de la base de datos por lo que es necesario consultar los archivos de documentación de los controladores.

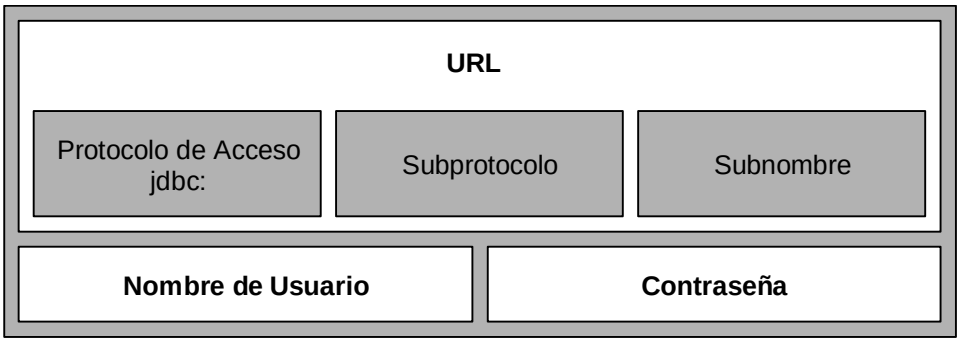

Figura 3.26. Formato de la Cadena de Conexión.

En (Matthews, 2004; Microsoft SQL Server Development Team, 2002; Sanko, et. al., 2001) se encuentra la documentación relacionada de los valores necesarios para conectar la "Interfaz Común SQL" con los motores de base de datos elegidos.

El Cuadro 3.3 muestra el formato de las URL que se utilizan para la conexión de los programas basados en JDBC, sin embargo los nombres de las bases de datos locales a acceder, los nombres de usuario válidos con sus contraseñas, deben de ser proporcionados por el DBA de la base de datos distribuida. Importante: los parámetros de la URL deben ocupar solo un renglón, es decir, no deben existir espacios en blanco o saltos de línea entre cada elemento.

También, es primordial tener en cuenta que los nombres de usuario que se utilicen deben tener privilegios de conexión y de escritura sobre los datos de las tablas locales a las que se desea consultar o manipular de manera remota.

Cuadro 3.3. Formato de URL para conexión a los DBMS elegidos.

NOMBRE DEL DBMS	FORMATO DE URL
Microsoft SQL Server 2000 Desktop Edition	jdbc:microsoft:sqlserver://<Dirección IP>:<Puerto>; databasename=<Nombre de base de datos local>; user=<Nombre de usuario>; password=<Contraseña>
MySQL Database Server 4.0.20	jdbc:mysql://<Dirección IP>:<Puerto>/ <Nombre de base de datos local>? user=<Nombre de usuario>& password=<Contraseña>
Oracle 9i Database Server (9.0.1)	jdbc:oracle:thin: <Nombre de usuario>/ <Contraseña>@ <Dirección IP>: <Puerto>: <Nombre de base de datos local>

Una vez configurado correctamente el archivo "servidores.txt" se procede a utilizarlo cuando se requiera iniciar una conexión a uno o a todos los DBMS para poder ejecutar en ellos las sentencias SQL de obtención y/o manipulación de datos generados por el módulo "Generador de consultas"". Las actividades relacionadas a este caso de uso se muestran en la Figura 3.27.

III.3.5.2. Caso de uso Abrir conexión.

Esta sección es utilizada por el módulo "Comunicaciones" para iniciar las rutinas de conexión a los servidores de la base de datos distribuida. Ésta se realiza utilizando las rutinas de Java necesarias y los parámetros globales JDBC obtenidos en el caso de uso anterior.

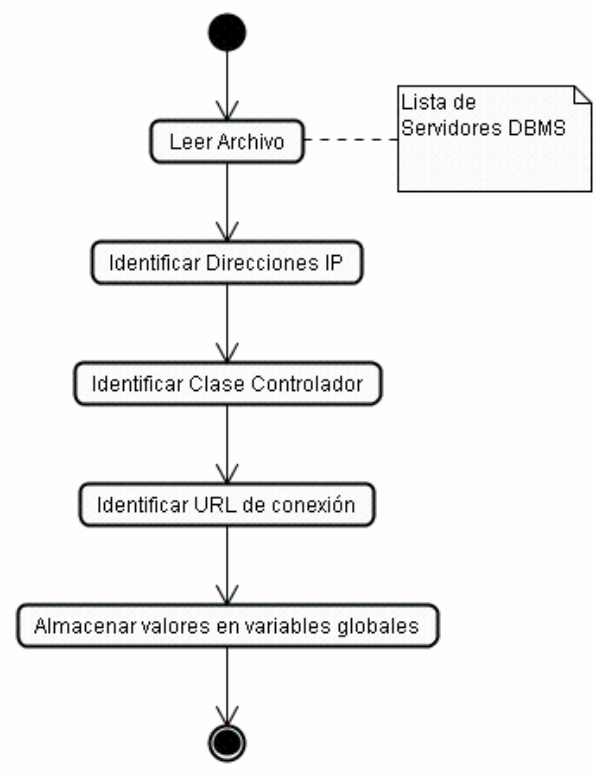

Figura 3.27. Actividades para obtener los Servidores DBMS.

Las actividades de este caso de uso (ver Figura 3.28) son:

- Utilizando los atributos de la clase, se obtienen los parámetros necesarios para abrir una conexión, dependiendo de la dirección IP del servidor en donde se requiera ejecutar una sentencia SQL generada anteriormente.
- Con la programación adecuada se orienta al módulo “Comunicaciones” a conectarse usando la clase *driver* del DBMS y el URL de conexión adecuado.
- Al resultar la conexión sin errores, Java crea un objeto del tipo *Connection* para mantenerla residente en la memoria del sistema y que se pueda seguir utilizando.
- Si existe un error en este proceso pues se envía un mensaje de estado del Servidor elegido a la “Interfaz Común SQL” para que tome las acciones adecuadas para informarle a las aplicaciones Cliente.

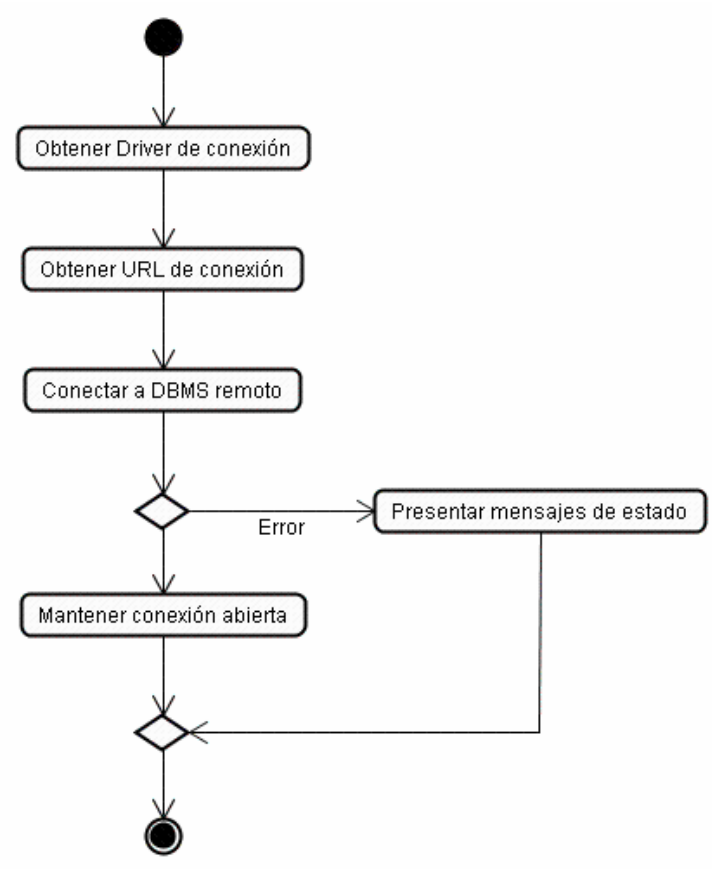

Figura 3.28. Actividades para abrir una conexión remota.

III.3.5.3. Caso de uso Ejecutar instrucciones SQL.

Esta sección la constituyen dos métodos de la clase “Comunicaciones” que se muestran en la Figura 3.10. El uso de éstos es necesario porque las rutinas Java para ejecutar las instrucciones SQL son diferentes.

Las instrucciones de manipulación SQL: “DELETE”, “INSERT” y “UPDATE” tienen la particularidad de que no devuelven un conjunto de datos al terminar de ser ejecutadas, únicamente regresan el control a la clase que las ejecutó y un mensaje acerca del estado en que se encuentra la base de datos después de ser modificada por éstas. Ver Figura 3.29.

La instrucción de obtención o consulta SQL: “SELECT” es la única que devuelve un conjunto de datos relacionados a través de los valores de búsqueda definidos en sus

cláusulas. Este conjunto, conocido en Java como *ResultSet* es un tipo de dato especial que se comporta como un vector de datos de tamaño *s*, donde cada elemento del mismo contiene una fila de la tabla en la que fue ejecutada la instrucción.

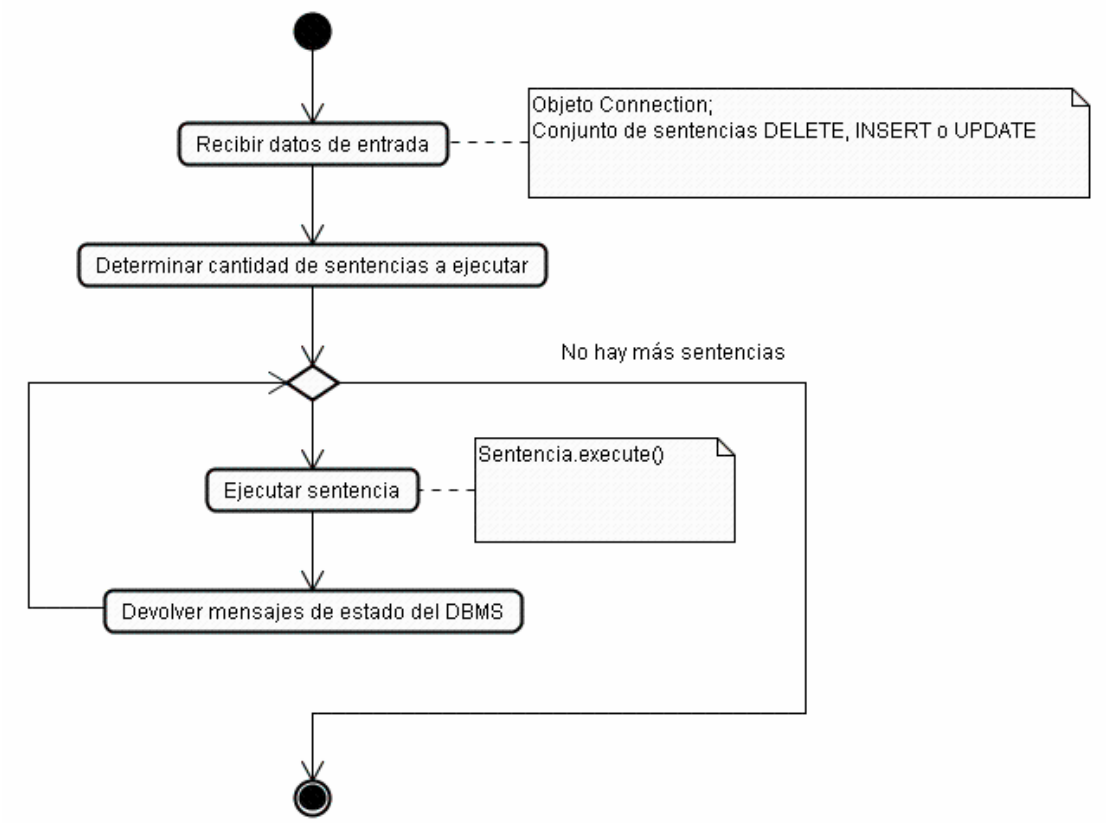

Figura 3.29. Actividades para ejecutar sentencias de manipulación de datos.

En el caso de que el *ResultSet* no contenga ninguna información a devolver, éste obtiene el valor nulo (`NULL`). Ver Figura 3.30.

III.3.5.4. Caso de uso Cerrar conexión.

Por último, esta sección de la clase "Comunicaciones" se encarga únicamente de cerrar la conexión creada de cada uno de los motores relacionales que soportan a la base de datos distribuida. Requiere como dato de entrada, el objeto *Connection* creado en el caso de uso Abrir conexión. Ver Figura 3.31.

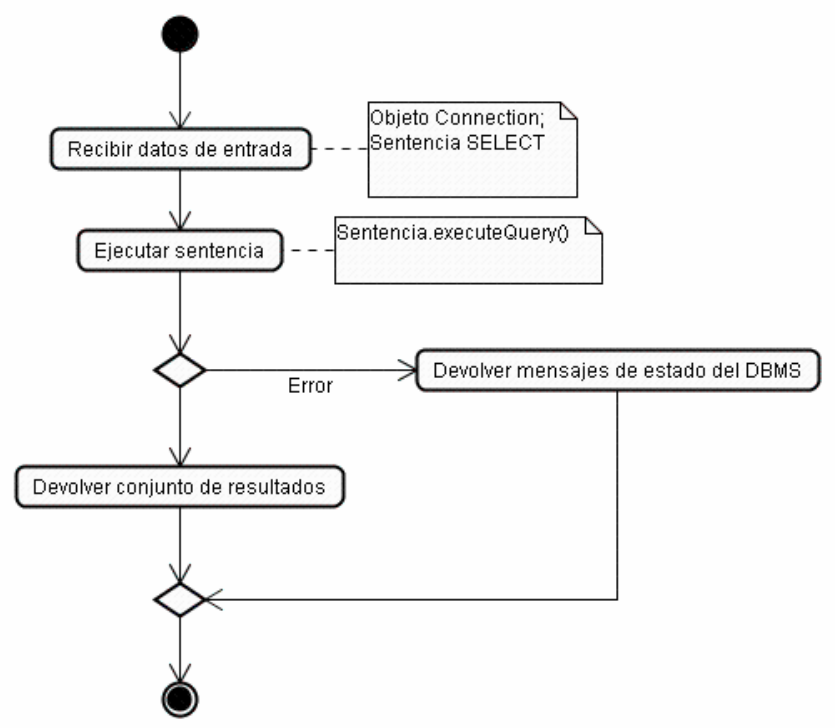

Figura 3.30. Actividades para ejecutar sentencias de obtención de datos.

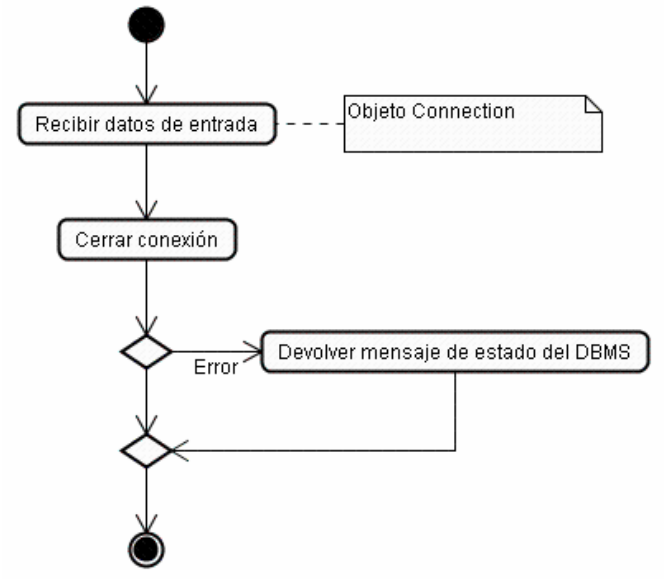

Figura 3.31. Actividades para cerrar una conexión remota.

III.4. Recomendaciones para el control de la Heterogeneidad.

El modelo Relacional de datos no define los tipos de datos que cada uno de los atributos de cierta relación tiene que poseer para representar correctamente la información que almacenará. El Lenguaje para Base de Datos SQL a lo largo de sus versiones, define ciertos tipos de datos que ayudan al programador a poder construir bases de datos que se apeguen a la realidad del entorno en el cual se requieren ejecutar. En la sección II.2.2 “Servidores de base de datos” se muestran los tipos de datos que cada una de las versiones de SQL soportan para tal fin y, en los Cuadros 2.1, 2.2 y 2.3 de la sección II.3.2 “El lenguaje de programación Java y bases relacionales de datos” se concentra la equivalencia de los tipos de datos SQL con los tipos de datos que Java utiliza dentro de sus programas. Sin embargo, en los ambientes de bases de datos distribuidas y heterogéneas, esto no es tan sencillo como parece.

El problema de los ambientes operativos que utilizan base de datos distribuida recae precisamente en los motores relacionales de cada sitio de la red que la integra. Cada DBMS utiliza, al igual que las versiones del lenguaje SQL que soportan (sin dejar de tomar en cuenta las modificaciones en la funcionalidad de cada una de ellas), su propio conjunto de tipos de datos para representarlos. Esto hace que la información se almacene de distintas maneras, en ocasiones imperceptibles, pero que pueden en algún momento causar problemas al momento de obtenerlos, manipularlos y presentarlos en las interfaces de usuario.

Aunado a lo anterior, también es necesario tener en cuenta los tipos de datos que utilizan las aplicaciones Cliente para capturar la información de un usuario. Un *front – end* creado usando Java necesita que cada variable que almacene en memoria los valores introducidos tenga un tipo de dato. Si fuese hecho en C tendría otro e incluso, sólo un “tipo” (aunque no se le declare explícitamente) si la GUI estuviese basada en HTML que únicamente maneja los datos obtenidos de un formulario como cadenas de texto. El Campo de acción experimental utiliza este entorno operativo con el fin de proponer alternativas de solución a estos inconvenientes.

La “Interfaz Común SQL” fue diseñada para manejar la sintaxis de SQL-92 y por lo tanto el uso de los tipos de datos que son utilizados por esta versión es inherente, sin

embargo, dos de los DBMS elegidos utilizan SQL-99 para representar a sus datos e incluso Oracle usa un tipo de datos “propio” para almacenar los caracteres. La compatibilidad entre ellos se puede resolver utilizando la información proporcionada en el Cuadro 3.4.

En la actualidad la mayoría de las aplicaciones Cliente están basadas en tecnología Internet por lo que las interfaces de usuario están construidas con HTML. Al capturar estos datos, las páginas Web los interpretan como simples cadenas de texto aunque sean fechas o cantidades numéricas. Para que un DBMS remoto pueda guardar estos valores es necesario convertirlos antes al formato de un tipo de dato que comprenda la base de datos, si no se realiza este procedimiento, una fecha por ejemplo: 28 de enero del año 2005 (‘28/01/2005’) será interpretada de diferentes maneras, al igual que el valor numérico 0.1 dependiendo del motor relacional que almacenará esa información, vea la Figura 3.32.

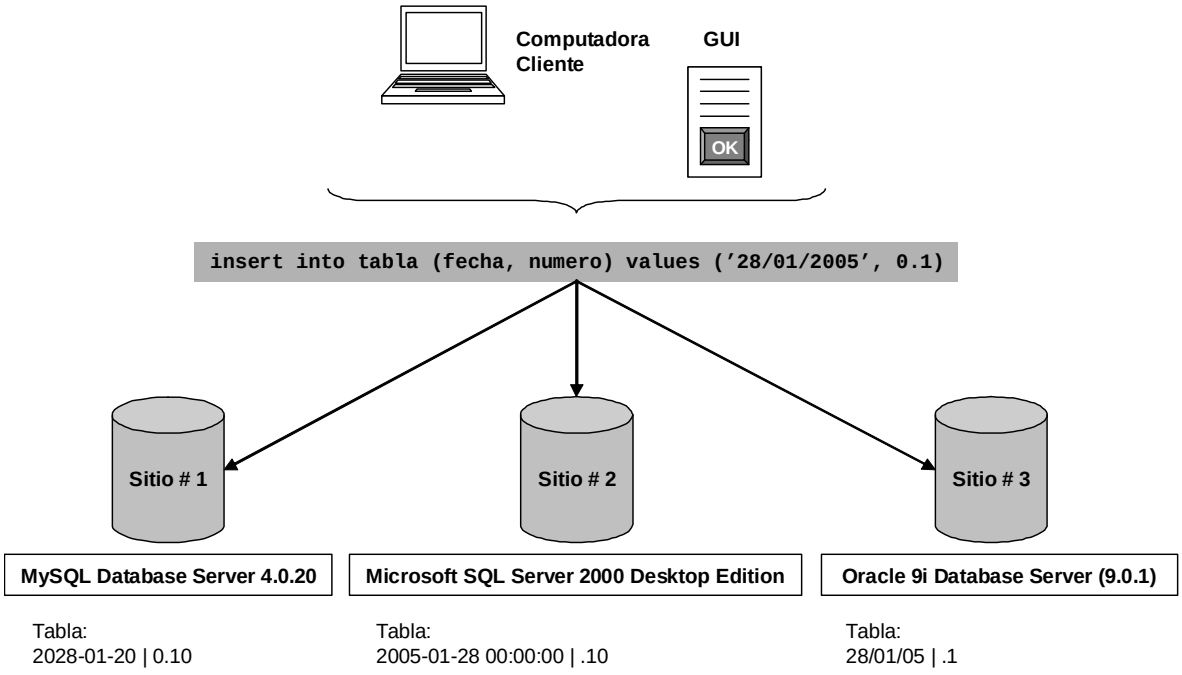

Figura 3.32. Formas de almacenar información en los DBMS elegidos.

Como se aprecia en la Figura 3.32, los valores que se introducen en los DBMS del Campo de acción experimental sufren un cambio tanto en formato como en interpretación.

Cuadro 3.4. Equivalencias entre tipos de datos de los DBMS elegidos.

SQL-92	MySQL Database Server 4.0.20	Microsoft SQL Server 2000 Desktop Edition	Oracle 9i Database Server (9.0.1)
CHARACTER	CHAR	CHAR	CHAR
CHAR			
CHARACTER VARYING	VARCHAR	VARCHAR	VARCHAR2
CHAR VARYING			
VARCHAR			
NATIONAL CHARACTER	CHAR, sólo que la base de datos utiliza el juego de caracteres predeterminado.	NCHAR	NCHAR
NATIONAL CHAR			
NCHAR			
NATIONAL CHARACTER VARYING	VARCHAR, sólo que la base de datos utiliza el juego de caracteres predeterminado.	NVARCHAR	NVARCHAR2
NATIONAL CHAR VARYING			
NCHAR VARYING			
NUMERIC	DECIMAL	NUMERIC	NUMBER
DECIMAL		DECIMAL	
DEC			
INTEGER	INT	INT	NUMBER, con un valor de precisión de 38 y con la escala igual a 0.
INT			
SMALLINT	SMALLINT	SMALLINT	
FLOAT	FLOAT, no se requiere indicar una precisión.	FLOAT, con un valor de precisión de 53.	NUMBER. La máxima precisión es de 38 y escala variable.
REAL	DOUBLE	REAL	NUMBER. Con precisión de 18 y escala variable.
DOUBLE PRECISION		FLOAT, con un valor de precisión de 24.	NUMBER con precisión 38 y escala variable.
BIT	No aplicable.	binary	No aplicable.
BIT VARYING		var binary	
DATE	DATE en formato 'YYYY-MM-DD'.	datetime o smalltime	DATE, sólo que depende de la configuración del idioma del DBMS. En español es 'DD/MM/YY'
TIME	TIME en formato 'HH:MM:SS'. No aplicable si especifica algún parámetro de SQL-92.	datetime o smalltime, sólo si almacena el valor en formato 'HH:MM:SS:ms'	No aplicable.
TIMESTAMP	TIMESTAMP en formato 'YYYYMMDDHHMMSS'. No aplicable si especifica algún parámetro de SQL-92.	timestamp	TIMESTAMP. Admite el parámetro WITH TIME ZONE de SQL-92.
INTERVAL	No aplicable.	No aplicable.	No aplicable.

El valor de la columna fecha es interpretado erróneamente por MySQL Database Server 4.0.20, en Microsoft SQL Server 2000 Desktop Edition el valor tiene por separadores un guión (-) y el formato es Año-Mes-Día con la añadidura de los campos Hora:Minuto:Segundo del tipo de datos *datetime* y únicamente Oracle 9i Database Server (9.0.1) lo hace más apegado al valor introducido. En el caso del valor numérico, únicamente cambia su formato. Si estos valores son utilizados en un programa Java, el valor de la fecha indicaría un error en la ejecución del mismo porque para que una variable tipo java.sql.Date lo almacene, éste necesita ser convertido en un dato tipo Long. Sin embargo, Long no admite caracteres como la diagonal (/), solo números. En el caso del valor numérico, BigDecimal lo interpreta de la manera correcta, pero con el formato “0.1” y para obtenerlo, el número introducido debe ser convertido a una cadena de caracteres porque en caso contrario, se indicaría un error.

Existe otro inconveniente. Los tipos de datos numéricos en los DBMS tienen una capacidad limitada para almacenar valores. El estándar SQL-92 no define la longitud máxima que un número pueda tener, a diferencia de las especificaciones técnicas que tienen los motores relacionales. En el Cuadro 3.4 por ejemplo, se muestra que un valor FLOAT definido en SQL-92 cabe de manera adecuada en un valor FLOAT de MySQL sin necesidad de especificar una precisión. Sin embargo, para que este valor pueda ser almacenado en Microsoft SQL Server, es necesario que se defina una precisión máxima de 53, si es mayor este valor, la cantidad a almacenar se vería truncada o redondeada dependiendo de las políticas del fabricante con respecto a la sobrecarga u *overflow* en una variable. En Oracle 9i este tipo de valor debe de almacenarse en una columna tipo NUMBER con precisión máxima de 38.

Otro ejemplo lo constituye el tipo de dato TIME de SQL-92, que en MySQL se utiliza el tipo TIME con el formato HH:MM:SS, mientras que en Microsoft SQL Server no existe con ese nombre, es necesario utilizar el tipo de datos propietario llamado *datetime* y añadirle el campo de milisegundos (ms). Por último, Oracle 9i no lo admite. Estos problemas en la capacidad de los tipos de datos que los DBMS pueden contener hacen que tenga que tenerse mucho cuidado al momento de intercambiar datos entre ellos.

En la búsqueda de una configuración óptima en el reparto de procesos a ejecutar en las capas Cliente, *middleware* y Servidor además de haber necesitado encontrar una

técnica que ayude a apaliar los inconvenientes anteriores, se recomienda realizar una conversión de datos desde la capa Cliente utilizando los mecanismos y herramientas Web de Java como *servlets* o la tecnología JSP (*Java Server Pages*) de este lenguaje de programación para conseguirlo y enlazarla con el *middleware* "Interfaz Común SQL" que se dedica a la comunicación e interacción con los Servidores que integran a una base de datos distribuida en un entorno de federación, a través de los Controladores JDBC y la generación de instrucciones de obtención y/o manipulación de datos.

Este tipo de especificación contempla que para conseguir una aplicación integrada, distribuida y heterogénea se abarquen las siguientes recomendaciones:

- Crear un esquema Conceptual de la base de datos a distribuir con base a los tipos de datos SQL-92.
- Realizar los procesos de fragmentación y de distribución del esquema conceptual global. Los esquemas resultantes serán los esquemas lógico – local de cada sitio en la red.
- En cada motor de base de datos a utilizar, crear cada esquema lógico – local utilizando los tipos de datos compatibles con SQL-92. En el caso de los DBMS elegidos utilice la equivalencia mostrada en el Cuadro 3.4.
- Concentre el esquema Conceptual, de Fragmentación y de Reparto en los archivos de configuración externos de la "Interfaz Común SQL". Vea el Apéndice A.2 para mayor información.
- Diseñe una interfaz de usuario basada en el Web de manera tal que se trate de cumplir ciertas consideraciones que debe tener una base de datos distribuida con vistas al usuario final. Vea la sección II.2.1 "Base de datos distribuidas".
- Realice los procesos de conversión y estandarización de formatos de los valores capturados. Almacénelos en variables que utilicen los tipos de datos Java compatibles con SQL-92 y los DBMS elegidos, que se muestran en el Cuadro 3.5.
- Enlace la aplicación Cliente con la "Interfaz Común SQL". Utilice los métodos públicos de la clase Control que se muestran en la Figura 3.10 y proporciónele los datos que requieran cada una de las instrucciones de

obtención y/o manipulación de datos. Con ellos, se crearán las sentencias SQL que serán ejecutadas en los Servidores que soportan la base de datos distribuida.

- Realice una nueva conversión en el formato y en los datos obtenidos de cada uno de los Servidores de la base de datos distribuida para tener una presentación uniforme de los mismos en la interfaz de usuario.
- Asegúrese de instalar los DBMS elegidos y de crear cuentas de usuario locales con privilegios de acceso y manipulación de las tablas que administra para las conexiones remotas.
- Proporcione a la "Interfaz Común SQL" los controladores JDBC necesarios para los Servidores de base de datos que desee utilizar. Por propósitos de investigación, se soporta a los DBMS que se muestran en el Campo de acción experimental de la Figura 3.2. También, de los parámetros de conexión de cada uno de ellos en el archivo "servidores.txt". Consulte el Apéndice A.2 para mayor información.
- Coloque la "Interfaz Común SQL", los Servidores que soportan a la base de datos distribuida y la aplicación Cliente en una arquitectura similar a la que se muestra en las Figuras 3.2 y 3.8.

Las aplicaciones Cliente basadas en HTML capturan la información introducida como cadenas de texto y aquellas que fueron construidas con Java, lo hacen según el tipo de datos asignado a las variables que las almacenan en la memoria principal de la computadora. Irremediablemente, al utilizar este lenguaje de programación para enviar a los Servidores que soportan a la base de datos distribuida esos datos, éstos se manipulan como anteriormente se menciona con la diferencia de que SQL, requiere que los valores que serán almacenados en columnas de tipo carácter, tiempo y fecha estén delimitados por un par de comillas simples (‘) y los que se dirigen a columnas numéricas no. Vea la Figura 3.32, los valores a agregar por la sentencia "INSERT" cumplen esta condición. Si no se tiene contemplada esta circunstancia, los DBMS indicarían un error en el proceso porque éstos requieren que ciertos valores estén delimitados.

A través de las equivalencias entre los tipos de datos de los DBMS participantes, de SQL-92 y de Java que se muestran en el Cuadro 3.5 se pueden realizar aplicaciones distribuidas muy compatibles y uniformes en entornos operativos heterogéneos, evitando el proceso de truncar o redondear datos que realizan los motores de base de datos, los problemas de interpretación de los mismos hacia los valores y los diferentes formatos de presentación que los Servidores usan para ellos.

Cuadro 3.5. Equivalencias de tipos de datos de los DBMS elegidos y Java.

TIPO DE DATO PRIMITIVO O POR REFERENCIA DE JAVA	TIPO DE DATO DE SQL-92 Y DBMS
short	SMALLINT
int o java.lang.Integer	INTEGER
	INT
float o java.lang.Float	REAL
double o java.lang.Double	DOUBLE PRECISION
char	NATIONAL CHARACTER
java.lang.String	CHARACTER VARYING
	CHAR VARYING
	VARCHAR
java.math.BigDecimal	NUMERIC
java.lang.Boolean	BIT
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP

IV. RESULTADOS Y DISCUSIÓN

La “Especificación de una Interfaz Común SQL para la comunicación entre bases de datos heterogéneas en diferentes sistemas operativos” resalta una problemática muy común en los Sistemas de Bases de Datos en Federación. Las diferencias entre las versiones del Lenguaje para Bases de Datos SQL que utilizan los DBMS aunado a los diversos puntos de vista que los fabricantes de estos sistemas tienen acerca de la representación de los datos, la eficiencia y el rendimiento de un producto con el fin de incrementar la productividad de los usuarios finales y obtener más cuota de mercado, hacen que la integración y elección de diferentes tecnologías que soportan a un sistema distribuido sea un proceso complicado.

La estructura de esta investigación proporciona una visión general con tendencia a lo particular sobre la forma en que los sistemas distribuidos y heterogéneos de base de datos son integrados. Muestra además, cómo la “Interfaz Común SQL” es funcional dentro de un ambiente Cliente/Servidor y de cómo interactúan sus componentes principales y externos para crear la “ilusión de una base de datos única”, apreciada por el usuario final a través de una interfaz de usuario.

El seguir las recomendaciones para el “Control de Heterogeneidad” vistas en la sección III.4 del capítulo anterior en conjunción con la “Interfaz Común SQL”, las tecnologías para el despliegue de aplicaciones en Internet, el lenguaje de programación Java, los Controladores JDBC y de las equivalencias entre los tipos de datos del Cuadro 3.5 ayuda a aminorar el impacto de este inconveniente y que se puedan ejecutar sentencias SQL de obtención y/o manipulación de datos de manera remota a diferentes motores relacionales que soportan a una base de datos distribuida.

Como todo proyecto de investigación, el prototipo expuesto está sujeto a mejoras y ampliaciones. Por ejemplo, actualmente se ofrece soporte para los DBMS seleccionados y que conforman el Campo de acción experimental. Con la añadidura de Controladores JDBC especialmente diseñados para otros Sistemas de Administración de Base de Datos distintos a los integrados en la “Interfaz Común SQL” le permitirán a éste, realizar operaciones de obtención y/o manipulación de datos en ellos. Por otro lado, es posible ampliar la sintaxis de cada una de las instrucciones SQL-92 elegidas para conseguir búsquedas de información

más especializadas o generar de una manera más detallada, las operaciones de manipulación de datos.

El siguiente nivel en la evolución de una “Interfaz Común SQL” es expandirla hasta transformarla en una “Pasarela o Gateway Común SQL”, como se sugiere en (Orfali, et. al., 1998). Un sofisticado sistema cuya función es la de tener un solo controlador de conexión de tal manera que no necesite tener uno para cada motor relacional involucrado en el Sistema de Base de Datos en Federación. Esta arquitectura hace que el sistema tenga su propia manera de enviar y recibir datos e información entre el nivel Cliente y los Servidores, además de mensajes de control, traduciendo éstos al modelo nativo u original de los DBMS que soportan a la base de datos distribuida. En la primera iteración esto no es posible porque se utiliza el Formato y Protocolo o FAP (*Format And Protocols*) de mensajes de los Controladores JDBC creado por cada fabricante; la finalidad es tener un controlador común que sea compatible con cada uno de éstos.

LITERATURA CITADA

Alarcón Cavero, Pedro Pablo. 1994. El estándar ANSI SQL 86. Primera edición. Universidad Politécnica de Madrid. España.

Booch, Grady; Rumbaugh, James; Jacobson, Ivar. 1999. El Lenguaje Unificado de Modelado. Primera edición. Editorial Addison Wesley Iberoamericana. España, p. 79-102.

Castaño, Adoración de Miguel; Piattini Velthius, Mario G. 1999. Fundamentos y modelos de bases de datos. Segunda edición. Editorial Alfaomega Grupo Editor, S. A de C. V. México, p. 265-296.

Ceballos Sierra, Francisco Javier. 2000. Java 2. Curso de programación. Primera edición. Editorial Alfa Omega Grupo Editor, S. A. de C. V. México.

Date, C. J. 2001. Introducción a los sistemas de bases de datos. Séptima edición. Editorial Pearson Educación de México, S. A. de C. V. México, p. 1-57, 150-197, 651-693.

Elmasri, Ramez; Navathe, Shamkant B. 2000. Sistemas de bases de datos. Segunda edición. Editorial Addison Wesley Longman de México, S. A. de C. V. México, p. 1-67, 456-486, 704-729.

International Standards Organization, International Electrotechnical Comision. 2005. Final Committee Draft of ISO/IEC 9075-2:2003. Primera edición. Estados Unidos de América

Froufe Quintas, Agustín. 2000. Java 2. Manual de usuario y tutorial. Segunda edición. Alfaomega Grupo Editor, S. A. de C. V. México.

Gosling, James; Joy, Hill; Steele, Guy. 1996. The Java Language Specification. Primera edición. Editorial Addison Wesley Longman. Estados Unidos de América.

Kendall Kenneth E.; Kendall Julie E. 1997. Análisis y diseño de sistemas. Tercera edición. Editorial Prentice Hall Iberoamericana, S. A. México, p. 27-48.

Matthews, Mark. 2004. MySQL Connector/J Documentation. MySQL AB. Suecia.

Melton, Jim; Eisenberg, Andrew. 2002. SQL y Java. Guía para SQLJ, JDBC y tecnologías relacionadas. Alfaomega Grupo Editor, S. A. de C. V. México.

Melton, Jim; Simon, Alan. 1993. Understanding the New SQL: A Complete Guide. Editorial Morgan Kaufmann. Estados Unidos de América.

Melton, Jim; Simon, Alan. 2001. SQL 1999 - Understanding Relational Language Components. Editorial Morgan Kaufmann. Estados Unidos de América.

Microsoft SQL Server Development Team. 2002. Microsoft SQL Server 2000 Driver for JDBC – User's guide and reference. Microsoft Corporation. Estados Unidos de América.

Orfali, Robert; Harkey, Dan; Edwards, Jeri. 1998. Cliente/Servidor. Guía de supervivencia. Segunda edición. Editorial McGraw-Hill Interamericana Editores S. A. de C. V. México, p. 23-32, 99-112, 149-198.

Pressman, Roger S. 2002. Ingeniería de *Software*. Un enfoque práctico. Quinta edición. Editorial McGraw-Hill/Interamericana de España, S. A. U. España, p. 245-314.

Real Academia Española. 2001. Diccionario de la Lengua Española. Vigésima segunda edición. Dirección de Internet: http://www.rae.es

Sanko, Mike; Wright, Brian; Pfaeffle, Thomas. 2001. Oracle 9i JDBC Developer's guide and reference. Oracle Corporation. Estados Unidos de América.

Silberschatz, Abraham; Korth, Henry F.; Sudarshan S. 1998. Fundamentos de bases de datos. Tercera edición. Editorial McGraw-Hill Interamericana de España, S. A. U. España.

Stallings, William. 1997. Comunicaciones y redes de computadores. Quinta edición. Editorial Prentice Hall Iberia, España.

Tanenbaum, Andrew S. 1997. Redes de computadoras. Tercera edición. Editorial Prentice Hall Hispanoamericana, S. A. México.

APÉNDICE

A.1. Código fuente del prototipo de la Interfaz Común SQL.

La programación de este componente *middleware* se realizó utilizando el lenguaje de programación Java versión 1.4.2_15 de la empresa Sun Microsystems. Para más detalles acerca del funcionamiento en conjunto de la Interfaz Común SQL es necesario consultar el capítulo III “Metodología” del presente documento.

Nombre de Clase: Archivo Descripción: Provee acceso a la lectura de los archivos que se le indiquen, almacenando su contenido en una arreglo de caracteres.

```
package icsql.clases;

import java.io.File;
import java.io.FileReader;

public class Archivo
{
  char[] leer(String sNombArch)
  {
    char cBuf[];
    File fRef;
    FileReader frLect;
    int iTam;
    Long lVal;

    cBuf = new char[1];
    try
    {
      fRef = new File(sNombArch);
      frLect = new FileReader(fRef);
      lVal = new Long(fRef.length());
```

```
      iTam = lVal.intValue();
      cBuf = new char[iTam];
      frLect.read(cBuf,0,iTam);
      frLect.close();
    }
    catch(Throwable t)
    {
      System.out.println(t.getMessage());
    }
    return(cBuf);
  }
}
```

Nombre de Clase: Comunicaciones Descripción: Proporciona las operaciones relacionadas a la conexión, acceso y ejecución de sentencias SQL a una base de datos indicada utilizando JDBC.

```
package icsql.clases;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.StringTokenizer;
import java.util.Vector;

public class Comunicaciones
{
  int iPosNombSitio;
  Vector vControl, vNombSitio, vParam;

  Connection abrirConexion(String sNombSitio)
  {
    boolean bErr;
    Connection cCon;

    bErr = false;
```

```
  cCon = null;
  obtenerServidores();
  bErr = buscarConfiguracion(sNombSitio);
  if(bErr == false)
  {
    try
    {
      Class.forName(vControl.elementAt(iPosNombSitio).toString());
      cCon =  DriverManager.getConnection(vParam.
        elementAt(iPosNombSitio).toString());
    }
    catch(Throwable t)
    {
      System.out.println(t.getMessage());
    }
  }
  return(cCon);
}

boolean buscarConfiguracion(String sNombSitio)
{
  boolean bErr;
  int i;

  bErr = false;
  iPosNombSitio = -1;
  for(i = 0; i < vNombSitio.size(); i++)
  {
    if(sNombSitio.compareTo(vNombSitio.elementAt(i).toString()) == 0)
    {
      iPosNombSitio = i;
    }
  }
  if(iPosNombSitio < 0)
  {
    bErr = true;
  }
  return(bErr);
```

```
}

void cerrarConexion(Connection cCon)
{
  try
  {
    cCon.close();
  }
  catch(Throwable t)
  {
    System.out.println(t.getMessage());
  }
}

boolean ejecutar(Connection cCon, Vector vCons)
{
  boolean bResEjec;
  int i;
  Statement sInstr;

  bResEjec = false;
  sInstr = null;
  try
  {
    for(i = 0; i < vCons.size(); i++)
    {
      sInstr = cCon.createStatement();
      sInstr.execute(vCons.elementAt(i).toString());
      sInstr.close();
    }
    bResEjec = true;
  }
  catch(Throwable t)
  {
    System.out.println(t.getMessage());
  }
  return(bResEjec);
}
```

```
ResultSet ejecutarConsulta(Connection cCon, String sCons)
{
  ResultSet rsResEjec;
  Statement sInstr;

  rsResEjec = null;
  try
  {
    sInstr = cCon.createStatement();
    rsResEjec = sInstr.executeQuery(sCons);
  }
  catch(Throwable t)
  {
    System.out.println(t.getMessage());
  }
  return(rsResEjec);
}

void obtenerServidores()
{
  Archivo objArch;
  char cBuf[];
  int i, j, iTamArch;
  String sCont, sLin;
  StringTokenizer st1, st2;

  objArch = new Archivo();
  cBuf = objArch.leer("C:\\Tesis\\icsql\\esquemas\\servidores.txt");
  sCont = new String(cBuf);
  st1 = new StringTokenizer(sCont, "\n\r");
  iTamArch = st1.countTokens();
  vControl = new Vector();
  vNombSitio = new Vector();
  vParam = new Vector();
  for(i = 0; i < iTamArch; i++)
  {
    sLin = st1.nextToken();
```

```
      if(!sLin.startsWith("REM"))
      {
        st2 = new StringTokenizer(sLin, "*");
        vNombSitio.addElement(st2.nextToken().trim());
        vControl.addElement(st2.nextToken().trim());
        vParam.addElement(st2.nextToken().trim());
      }
    }
  }
}
```

Nombre de Clase: Control Descripción: Clase principal de la Interfaz Común SQL y que es el punto de partida para la utilización del resto de las clases. Es el punto intermedio entre la aplicación Cliente y las aplicaciones Servidor de los motores de bases de datos.

```
package icsql.clases;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.util.Vector;

public class Control
{
  public String actualizar(String sNombSitio, String sNombTab, Vector
    vColAct, Vector vColBusq, Vector vDatTab, Vector vValBusq)
  {
    boolean bErr;
    Comunicaciones objCom;
    Connection cCon;
    Generador objGen;
    int i;
    String sMens;
    Vector vCons;

    bErr = false;
```

```
  sMens = new String();
  objCom = new Comunicaciones();
  cCon = objCom.abrirConexion(sNombSitio);
  if(cCon != null)
  {
    objGen = new Generador();
    vCons = new Vector();
    vCons = objGen.updateStatement(sNombSitio, sNombTab, vColAct,
      vColBusq, vDatTab, vValBusq);
    bErr = objCom.ejecutar(cCon, vCons);
    objCom.cerrarConexion(cCon);
    if(bErr)
    {
      sMens = "Los datos han sido almacenados exitosamente.";
    }
    else
    {
      sMens = "No se permite el proceso de estos datos.";
    }
  }
  else
  {
    sMens = "Su operación no puede ser atendida en este momento, por
      favor inténtelo más tarde.";
  }
  return(sMens);
}

public String agregar(String sNombSitio, String sNombTab,
  Vector vDatTab)
{
  boolean bErr;
  Comunicaciones objCom;
  Connection cCon;
  Generador objGen;
  int i;
  String sMens;
  Vector vCons;
```

```
  bErr = false;
  sMens = new String();
  objCom = new Comunicaciones();
  cCon = objCom.abrirConexion(sNombSitio);
  if(cCon != null)
  {
    objGen = new Generador();
    vCons = new Vector();
    vCons = objGen.insertStatement(sNombSitio, sNombTab, vDatTab);
    bErr = objCom.ejecutar(cCon, vCons);
    objCom.cerrarConexion(cCon);
    if(bErr)
    {
      sMens = "Los datos han sido almacenados exitosamente.";
    }
    else
    {
      sMens = "No se permite el proceso de estos datos.";
    }
  }
  else
  {
    sMens = "Su operación no puede ser atendida en este momento, por
      favor inténtelo más tarde.";
  }
  return(sMens);
}

public String borrar(String sColBusq, String sNombSitio,
  String sNombTab, String sValBusq)
{
  boolean bErr;
  Comunicaciones objCom;
  Connection cCon;
  Generador objGen;
  int i;
  String sMens;
```

```
  Vector vCons;

  bErr = false;
  sMens = new String();
  objCom = new Comunicaciones();
  cCon = objCom.abrirConexion(sNombSitio);
  if(cCon != null)
  {
    objGen = new Generador();
    vCons = new Vector();
    vCons = objGen.deleteStatement(sColBusq, sNombSitio, sNombTab,
      sValBusq);
    bErr = objCom.ejecutar(cCon, vCons);
    objCom.cerrarConexion(cCon);
    if(bErr)
    {
      sMens = "Los datos han sido eliminados exitosamente.";
    }
    else
    {
      sMens = "No se permite el proceso de estos datos.";
    }
  }
  else
  {
    sMens = "Su operación no puede ser atendida en este momento, por
      favor inténtelo más tarde.";
  }
  return(sMens);
}

public String buscar(String sColBusq, String sNombTab, String sValBusq,
  Vector vNombSitio)
{
  Comunicaciones objCom;
  Connection cCon;
  Generador objGen;
  int i, iTotColRs, iTotErr, j;
```

```
String sCons, sMens;
ResultSet rs;
ResultSetMetaData rsmd;
Vector vCons, vRes;

objCom = new Comunicaciones();
objGen = new Generador();
iTotErr = 0;
sCons = new String();
sMens = new String();
vCons = new Vector();
for(i = 0; i < vNombSitio.size(); i++)
{
  sCons = objGen.querySpecification(sColBusq,
    vNombSitio.elementAt(i).toString(), sNombTab, sValBusq);
  if(sCons != null)
  {
    vCons.addElement(sCons);
  }
}
if(vCons.size() > 0)
{
  for(i = 0; i < vNombSitio.size(); i++)
  {
    cCon = objCom.abrirConexion(vNombSitio.elementAt(i).toString());
    if(cCon != null)
    {
      rs = objCom.ejecutarConsulta(cCon,
        vCons.elementAt(i).toString());
      try
      {
        rsmd = rs.getMetaData();
        iTotColRs = rsmd.getColumnCount();
        while(rs.next())
        {
          for(j = 1; j < iTotColRs + 1); j++)
          {
            sMens = sMens.concat(rs.getString(rsmd.getColumnName(j))
```

```
                + " | ");
            }
            sMens = sMens.concat("\n");
          }
          objCom.cerrarConexion(cCon);
        }
        catch(Throwable t)
        {
          System.out.println(t.getMessage());
          iTotErr++;
        }
      }
      else
      {
        iTotErr++;
      }
    }
    if(iTotErr == vNombSitio.size())
    {
      sMens = "#ICSQLERR-A";
    }
  }
  else
  {
    sMens = "#ICSQLERR-B";
  }
  return(sMens);
 }
}
```

Nombre de Clase: Generador Descripción: Accede a los esquemas conceptual, de fragmentación y de reparto de una base de datos distribuida y construye las sentencias INSERT, DELETE, SELECT y UPDATE del lenguaje SQL dependiendo de los datos de entrada enviados por la aplicación Cliente.

```
package icsql.clases;
```

```
import java.io.IOException;
import java.sql.Connection;
import java.util.StringTokenizer;
import java.util.Vector;

public class Generador
{
  Vector vColTab, vColFrag, vNombSitFrag, vFragTab;

  void conceptual(String sNombTab)
  {
     Archivo objArch;
     char cBuf[];
     int i, iTotColsDest, iTotLinArch;
     String sColsDest, sContArch, sLin, sTok;
     StringTokenizer st1, st2, st3;

     sColsDest = new String();
     sTok = new String();
     objArch = new Archivo();
     cBuf = objArch.leer("C:\\Tesis\\icsql\\esquemas\\conceptual.txt");
     sContArch = new String(cBuf);
     st1 = new StringTokenizer(sContArch, "\n\r");
     iTotLinArch = st1.countTokens();
     for(i = 0; i < iTotLinArch; i++)
     {
       sLin = st1.nextToken().trim();
       if(!sLin.startsWith("REM"))
       {
         st2 = new StringTokenizer(sLin, "=");
         sTok = st2.nextToken().trim();
         if(sTok.compareTo(sNombTab) == 0)
         {
           sTok = st2.nextToken().trim();
           sColsDest = sTok;
         }
       }
     }
```

```
   st3 = new StringTokenizer(sColsDest, "(,)");
   iTotColsDest = st3.countTokens();
   for(i = 0; i < iTotColsDest; i++)
   {
     sTok = st3.nextToken().trim();
     vColTab.addElement(sTok);
   }
}

Vector deleteStatement(String sColBusq, String sNombSitio,
  String sNombTab, String sValBusq)
{
  int i;
  String sCons;
  Vector vCons;

  iniciarGeneracion(sNombTab);
  vCons = new Vector();
  for(i = 0; i < vNombSitFrag.size(); i++)
  {
    if(sNombSitio.compareTo(vNombSitFrag.elementAt(i).toString()) == 0)
    {
      sCons = "DELETE FROM " + vFragTab.elementAt(i) + " WHERE " +
        sColBusq + "=" + sValBusq;
      vCons.addElement(sCons);
    }
  }
  return(vCons);
}

void fragmentacion(String sNombArch, String sNombTab)
{
  boolean bSal;
  int i;
  String sCols, sTok;
  StringTokenizer st;
  Tablas objTabl;
```

```
  bSal = false;
  objTabl = new Tablas();
  sCols = objTabl.obtenerColumnas(sNombArch, sNombTab.trim());
  st = new StringTokenizer(sCols, ",");
  do
  {
    if(sCols.trim().startsWith("(") && sCols.trim().endsWith(")"))
    {
      vFragTab.addElement(sNombTab.trim());
      vColFrag.addElement(sCols);
      bSal = true;
    }
    else
    {
      if(st.hasMoreTokens())
      {
        sTok = st.nextToken();
        fragmentacion(sNombArch, sTok.trim());
      }
      else
      {
        bSal = true;
      }
    }
  }while(!bSal);
}

void iniciarGeneracion(String sNombTab)
{
  vColTab = new Vector();
  vColFrag = new Vector();
  vNombSitFrag = new Vector();
  vFragTab = new Vector();
  conceptual(sNombTab);
  fragmentacion("C:\\Tesis\\icsql\\esquemas\\fragmentacion.txt",
    sNombTab);
  reparto();
}
```

```
Vector insertStatement(String sNombSitio, String sNombTab,
  Vector vDatTab)
{
  int i, iTotColsFrag, j, k;
  String sCol, sCons;
  StringTokenizer st;
  Vector vCons;

  vCons = new Vector();
  sCol = new String();
  iniciarGeneracion(sNombTab);
  for(i = 0; i < vNombSitFrag.size(); i++)
  {
    if(sNombSitio.compareTo(vNombSitFrag.elementAt(i).toString()) == 0)
    {
      sCons = new String("INSERT INTO ");
      sCons = sCons.concat(vFragTab.elementAt(i).toString() + " " +
        vColFrag.elementAt(i).toString() + " VALUES (");
      st = new StringTokenizer(vColFrag.elementAt(i).toString(),
        "(,)");
      iTotColsFrag = st.countTokens();
      for(k = 0; k < iTotColsFrag; k++)
      {
        sCol = st.nextToken().trim();
        for(j = 0; j < vColTab.size(); j++)
        {
          if(sCol.compareTo(vColTab.elementAt(j).toString()) == 0)
          {
            if(vDatTab.elementAt(j) != null)
            {
              sCons = sCons.concat(vDatTab.elementAt(j).toString());
            }
            else
            {
              sCons = sCons.concat("NULL");
            }
            if(k != (iTotColsFrag - 1))
```

```
            {
              sCons = sCons.concat(", ");
            }
            else
            {
              sCons = sCons.concat(")");
              vCons.addElement(sCons);
            }
          }
        }
      }
    }
  }
  return(vCons);
}

String querySpecification(String sColBusq, String sNombSitio,
  String sNombTab, String sValBusq)
{
  boolean bSal;
  int i, iPosFragTab, iPosFragTab1, iPosFragTab2, iTotColFrag, j, k;
  String sColCons, sColFrag, sCons, sNomCol1, sNomCol2;
  StringTokenizer st, st1, st2;
  Vector vPosFragTab;

  sColCons = new String();
  sColFrag = new String();
  sCons = new String();
  vPosFragTab = new Vector();
  iniciarGeneracion(sNombTab);
  for(i = 0; i < vFragTab.size(); i++)
  {
    if(sNombSitio.compareTo(vNombSitFrag.elementAt(i).toString()) == 0)
    {
      vPosFragTab.addElement(Integer.toString(i));
    }
  }
  i = 0;
```

```
while(i < vColTab.size())
{
  j = 0;
  while(j < vPosFragTab.size())
  {
    if(j > 0)
    {
      sColCons = sColCons.concat(", ");
    }
    iPosFragTab = new Integer(vPosFragTab.elementAt(j).toString()).
      intValue();
    st = new StringTokenizer(vColFrag.elementAt(j).toString(),
      "(,)");
    k = 0;
    iTotColFrag = st.countTokens();
    while(k < iTotColFrag)
    {
      sColFrag = st.nextToken().trim();
      if(vColTab.elementAt(i).toString().compareTo(sColFrag) == 0)
      {
        sColCons = sColCons.concat(vFragTab.elementAt(iPosFragTab).
          toString() + "." + sColFrag);
        if(k != (iTotColFrag - 1))
        {
          sColCons = sColCons.concat(", ");
        }
        i++;
      }
      k++;
    }
    j++;
  }
}
sCons = sCons.concat("SELECT " + sColCons + " FROM ");
for(i = 0; i < vPosFragTab.size(); i++)
{
  iPosFragTab = new Integer(vPosFragTab.elementAt(i).toString()).
    intValue();
```

```
  sCons = sCons.concat(vFragTab.elementAt(iPosFragTab).toString());
  if(i < (vPosFragTab.size() - 1))
  {
    sCons = sCons.concat(", ");
  }
}
bSal = false;
if(vPosFragTab.size() > 1)
{
  if(sValBusq.length() > 0)
  {
    sCons = sCons.concat(" WHERE ");
    i = 0;
    do
    {
      if(i < vPosFragTab.size())
      {
        iPosFragTab = new Integer(vPosFragTab.elementAt(i).
          toString()).intValue();
        st = new StringTokenizer(vColFrag.elementAt(iPosFragTab).
          toString(), "(,)");
        while(st.hasMoreTokens())
        {
          sColFrag = st.nextToken().trim();
          if(sColBusq.compareTo(sColFrag) == 0)
          {
            sCons = sCons.concat(vFragTab.elementAt(iPosFragTab).
              toString() + "." + sColBusq + "=" + sValBusq);
            bSal = true;
          }
        }
        i++;
      }
    }while(bSal == false);
    sCons = sCons.concat(" AND ");
    for(i = 0; i < vPosFragTab.size(); i++)
    {
      iPosFragTab1 = new Integer(vPosFragTab.elementAt(i).
```

```
      toString()).intValue();
    st1 = new StringTokenizer(vColFrag.elementAt(iPosFragTab1).
      toString(), "(,)");
    sNomCol1 = st1.nextToken().trim();
    for(j = i + 1; j < vPosFragTab.size(); j++)
    {
      iPosFragTab2 = new Integer(vPosFragTab.elementAt(j).
        toString()).intValue();
      st2 = new StringTokenizer(vColFrag.elementAt(iPosFragTab2).
        toString(), "(,)");
      while(st2.hasMoreTokens())
      {
        sNomCol2 = st2.nextToken().trim();
        if(sNomCol1.compareTo(sNomCol2) == 0)
        {
          sCons = sCons.concat(vFragTab.elementAt(iPosFragTab1).
            toString() + "." + sNomCol1 + "=" + vFragTab.elementAt
            (iPosFragTab2).toString() + "." + sNomCol2);
        }
      }
    }
  }
}
else
{
  sCons = sCons.concat(" WHERE ");
  for(i = 0; i < vPosFragTab.size(); i++)
  {
    iPosFragTab1 = new Integer(vPosFragTab.elementAt(i).
      toString()).intValue();
    st1 = new StringTokenizer(vColFrag.elementAt(iPosFragTab1).
      toString(), "(,)");
    sNomCol1 = st1.nextToken().trim();
    for(j = i + 1; j < vPosFragTab.size(); j++)
    {
      iPosFragTab2 = new Integer(vPosFragTab.elementAt(j).
        toString()).intValue();
      st2 = new StringTokenizer(vColFrag.elementAt(iPosFragTab2).
```

```
          toString(), "(,)");
        while(st2.hasMoreTokens())
        {
          sNomCol2 = st2.nextToken().trim();
          if(sNomCol1.compareTo(sNomCol2) == 0)
          {
            sCons = sCons.concat(vFragTab.elementAt(iPosFragTab1).
              toString() + "." + sNomCol1 + "=" + vFragTab.elementAt
              (iPosFragTab2).toString() + "." + sNomCol2);
          }
        }
      }
    }
  }
}
else
{
  if(sValBusq.length() > 0)
  {
    sCons = sCons.concat(" WHERE ");
    i = 0;
    do
    {
      if(i < vPosFragTab.size())
      {
        iPosFragTab = new Integer(vPosFragTab.elementAt(i).
          toString()).intValue();
        st = new StringTokenizer(vColFrag.elementAt(iPosFragTab).
          toString(), "(,)");
        while(st.hasMoreTokens())
        {
          sColFrag = st.nextToken().trim();
          if(sColBusq.compareTo(sColFrag) == 0)
          {
            sCons = sCons.concat(vFragTab.elementAt(iPosFragTab).
            toString() + "." + sColBusq + "=" + sValBusq);
            bSal = true;
          }
```

```
          }
          i++;
        }
      }while(bSal == false);
    }
  }
  return(sCons);
}

void reparto()
{
  Archivo objArch;
  char cBuf[];
  int i, j, k, iTotLin, iTotTok;
  String sContArch, sNombSitio, sLin, sTok;
  StringTokenizer st1, st2;

  sNombSitio = new String();
  vNombSitFrag = new Vector();
  objArch = new Archivo();
  cBuf = objArch.leer("C:\\Tesis\\icsql\\esquemas\\reparto.txt");
  sContArch = new String(cBuf);
  for(i = 0; i < vFragTab.size(); i++)
  {
    st1 = new StringTokenizer(sContArch, "\n\r");
    iTotLin = st1.countTokens();
    for(j = 0; j < iTotLin; j++)
    {
      sLin = st1.nextToken().trim();
      if(!sLin.startsWith("REM"))
      {
        st2 = new StringTokenizer(sLin, "=,");
        iTotTok = st2.countTokens();
        k = 0;
        while(k < iTotTok)
        {
          sTok = st2.nextToken().trim();
          if(k == 0)
```

```
          {
            sNombSitio = sTok;
          }
          else
          {
            if(sTok.compareTo(vFragTab.elementAt(i).toString()) == 0)
            {
              vNombSitFrag.addElement(sNombSitio);
            }
          }
          k++;
        }
      }
    }
  }
}

Vector updateStatement(String sNombSitio, String sNombTab,
  Vector vColAct, Vector vColBusq, Vector vDatTab, Vector vValBusq)
{
  boolean bSal;
  char cComa, cTmp[];
  Character chCarTmp, chComa;
  int i, iTamTmp, iTotTok, j, k;
  String sCol, sComa, sCons, sTmp;
  StringTokenizer st;
  Vector vCons;

  sComa = ",";
  cComa = sComa.charAt(0);
  chComa = new Character(cComa);
  iniciarGeneracion(sNombTab);
  sCons = new String();
  vCons = new Vector();
  for(i = 0; i < vNombSitFrag.size(); i++)
  {
    if(sNombSitio.compareTo(vNombSitFrag.elementAt(i).toString()) == 0)
    {
```

```
sCons = "UPDATE " + vFragTab.elementAt(i).toString() + " SET ";
st = new StringTokenizer(vColFrag.elementAt(i).toString(),
  "(,)");
iTotTok = st.countTokens();
for(j = 0; j < iTotTok; j++)
{
  sCol = st.nextToken().trim();
  bSal = false;
  k = 0;
  do
  {
    if(k < vColAct.size())
    {
      if(sCol.compareTo(vColAct.elementAt(k).toString()) == 0)
      {
        if(vDatTab.elementAt(k) != null)
        {
          sCons = sCons.concat(vColAct.elementAt(k).toString() +
            "=" + vDatTab.elementAt(k).toString() + ", ");
        }
        else
        {
          sCons = sCons.concat(vColAct.elementAt(k).toString() +
            "= NULL, ");
        }
        bSal = true;
      }
      k++;
    }
    else
    {
      bSal = true;
    }
  }while(bSal == false);
}
sTmp = new String(sCons.trim());
iTamTmp = sTmp.length();
cTmp = sTmp.toCharArray();
```

```
        chCarTmp = new Character(cTmp[(iTamTmp - 1)]);
        if(chCarTmp.compareTo(chComa) == 0)
        {
          sCons = new String(sTmp.substring(0, (iTamTmp - 1)));
        }
        if(vColBusq.size() > 1)
        {
          sCons = sCons.concat(" WHERE ");
          for(j = 0; j < vColBusq.size(); j++)
          {
            sCons = sCons.concat(vColBusq.elementAt(j).toString() + "=" +
              vValBusq.elementAt(j).toString());
            if(j < (vColBusq.size() - 1))
            {
              sCons = sCons.concat(" AND ");
            }
          }
        }
        else
        {
          sCons = sCons.concat(" WHERE " + vColBusq.elementAt(0).
            toString() + "=" + ValBusq.elementAt(0).toString());
        }
      vCons.addElement(sCons);
      }
    }
  return(vCons);
  }
}
```

Nombre de Clase: Tablas Descripción: Ubica las columnas de los fragmentos que componen a una tabla de la base de datos distribuida.

```
package icsql.clases;

import java.util.StringTokenizer;
```

```
public class Tablas
{
  String obtenerColumnas(String sNombArch, String sNombTab)
  {
    Archivo objArch;
    boolean bSal;
    char cBuf[];
    String sContArch, sLin, sTok;
    StringTokenizer st1, st2;

    bSal = false;
    sTok = new String();
    objArch = new Archivo();
    cBuf = objArch.leer(sNombArch);
    sContArch = new String(cBuf);
    st1 = new StringTokenizer(sContArch, "\n\r");
    do
    {
      sLin = st1.nextToken().trim();
      if(!sLin.startsWith("REM"))
      {
        st2 = new StringTokenizer(sLin, "=");
        sTok = st2.nextToken().trim();
        if(sTok.compareTo(sNombTab) == 0)
        {
          sTok = st2.nextToken().trim();
          bSal = true;
        }
      }
    }while(bSal == false);
    return(sTok);
  }
}
```

A.2. Componentes externos de la Interfaz Común SQL.

En el capítulo III “Metodología” y a lo largo de sus secciones se hizo referencia a los archivos de configuración que la “Interfaz Común SQL” utiliza para poder conectarse con los motores relacionales que dan soporte a la base de datos distribuida y conocer su estructura.

Nota: el formato para llenar cada archivo se explica en la sección III.3.4.1 “Caso de uso Obtener esquemas DB”.

Nombre de archivo: conceptual.txt

Descripción: Contiene el Esquema Conceptual y único que utiliza el Campo de acción experimental para verificar la funcionalidad de la “Interfaz Común SQL”.

```
departamento = (iddep, nombdep)
empleado = (curp, prinomb, segnomb, apepat, apemat, dir, codpos, sex,
  fecnac, salhr, depasig)
dirige = (emp, fecinigte, dep)
proyecto = (idproy, nombproy, depcontr)
trabaja = (emp, proy, feciniemp, fecfinemp)
```

Nombre de archivo: fragmentacion.txt

Descripción: Contiene el Esquema de Fragmentación que utiliza el Campo de acción experimental para verificar la funcionalidad de la “Interfaz Común SQL”.

```
departamento = dep1, dep2, dep3
dep1 = (iddep, nombdep)
dep2 = (iddep, nombdep)
dep3 = (iddep, nombdep)

empleado = emp1, emp2, emp3
emp1 = emp1per, emp1pro
emp2 = emp2per, emp2pro
emp3 = emp3per, emp3pro
```

```
emp1per = (curp, prinomb, segnomb, apepat, apemat, dir, codpos, sex,
  fecnac)
emp1pro = (curp, salhr, depasig)
emp2per = (curp, prinomb, segnomb, apepat, apemat, dir, codpos, sex,
  fecnac)
emp2pro = (curp, salhr, depasig)
emp3per = (curp, prinomb, segnomb, apepat, apemat, dir, codpos, sex,
  fecnac)
emp3pro = (curp, salhr, depasig)

dirige = dir1, dir2, dir3
dir1 = (emp, fecinigte, dep)
dir2 = (emp, fecinigte, dep)
dir3 = (emp, fecinigte, dep)

proyecto = proy1, proy2, proy3
proy1 = (idproy, nombproy, depcontr)
proy2 = (idproy, nombproy, depcontr)
proy3 = (idproy, nombproy, depcontr)

trabaja = trab1, trab2, trab3
trab1 = (emp, proy, feciniemp, fecfinemp)
trab2 = (emp, proy, feciniemp, fecfinemp)
trab3 = (emp, proy, feciniemp, fecfinemp)
```

Nombre de archivo: reparto.txt
Descripción: Contiene el Esquema de Reparto que utiliza el Campo de acción experimental para verificar la funcionalidad de la “Interfaz Común SQL”.

```
BaseDatos = dep1, emp1per, emp1pro, dir1, proy1, trab1
Programacion = dep2, emp2per, emp2pro, dir2, proy2, trab2
IngenieriaRequerimientos = dep3, emp3per, emp3pro, dir3, proy3, trab3
```

Por último, el archivo de configuración externo que se muestra a continuación fue descrito en la sección III.3.5.1 “Caso de uso Obtener Servidores DBMS”.

Nombre de archivo: servidores.txt
Descripción: Contiene los parámetros que utiliza la “Interfaz Común SQL” para poder conectarse con los motores relacionales que soportan a una base de datos distribuida.

```
BaseDatos * com.mysql.jdbc.Driver *
jdbc:mysql://192.168.0.14:3306/empresa?user=scott&
password=tiger

Programacion * com.microsoft.jdbc.sqlserver.SQLServerDriver *
jdbc:microsoft:sqlserver://192.168.0.21:1031;
databasename=empresa;user=scott;password=tiger

IngenieriaRequerimientos * oracle.jdbc.OracleDriver *
jdbc:oracle:thin:scott/tiger@192.168.0.28:1521:empresa
```