**UNIVERSIDAD NACIONAL**
**AUTÓNOMA DE MÉXICO**

**FACULTAD DE INGENIERÍA**

# DESARROLLO DE UN SISTEMA DE BASE DE DATOS DISTRIBUIDA EN UN CLUSTER DE ALTO DESEMPEÑO

## TESIS PROFESIONAL

QUE PARA OBTENER EL TÍTULO DE
INGENIERO EN COMPUTACIÓN

P R E S E N T A N:

CLAUDIA JAZMÍN DE LEÓN CRUZ
JOSÉ DE JESÚS NAVARRO CARRANZA


DIRECTORA DE TÉSIS
ING. LAURA SANDOVAL MONTAÑO

**CUIDAD UNIVERSITARIA**
**MÉXICO, D.F. 2006**

CLAUDIA JAZMIN DE LEÓN CRUZ
JOSÉ DE JESÚS NAVARRO CARRANZA

# DESARROLLO DE UN SISTEMA DE BASE DE DATOS DISTRIBUIDA EN UN CLUSTER DE ALTO DESEMPEÑO

JUNIO 2006

# *Agradecimientos*

Un agradecimiento a aquellas personas sin cuya participación no existiría este trabajo de tesis:

- A la Ing. Laura Sandoval Montaño por haber aceptado dirigir nuestro proyecto, por su confianza en nuestro trabajo, su apoyo y comprensión.

- Al Ing. Carlos Alberto Román Zamitiz por su orientación al inicio de nuestra investigación y por su invaluable apoyo en la realización de los últimos aspectos del proyecto.

- A la Ing. Elba Karén Sáenz García, responsable del Laboratorio de Telemática del Departamento de Ingeniería en Computación, por las facilidades que nos brindó para trabajar en el laboratorio.

- A nuestros profesores y compañeros de carrera, porque su enseñanza y colaboración se han reflejado directa e indirectamente en esta tesis.

*Claudia Jazmín de León Cruz*

*José de Jesús Navarro Carranza*

# *Agradecimientos*

Iniciar una obra es un acto relativamente asequible, pero perseverar en él y conseguir lo que se soñó, no lo es tanto; Se requiere de continuidad, esfuerzo y una fortaleza de espíritu, las cuales sólo se obtienen al rodearse con gente que influye en la vida propia, mostrándonos que no sólo existe un camino -sino varios-, pero que al final nos ayudarán a lograr nuestros anhelos y objetivos que nos hemos fijado en la vida.

Esta tesis refleja por lo que he luchado mucho tiempo, es el resultado de varios años de esfuerzo y dedicación. Pero no todo vino de mí, siempre estuve rodeada de seres que, sin saberlo, me ayudaron a llegar hasta este punto y a los cuales les estoy muy agradecida, pues sin ellos no estaría cerrando uno de los ciclos más importantes de mi vida, el cual servirá como base para seguir creciendo y ser un mejor ser humano.

En primer lugar quiero agradecer especialmente a mis padres, ***Josefina y David***, a quienes les debo la vida, los cuales han estado a mi lado en todas las facetas de ésta, apoyándome en todos los proyectos que he emprendido, quienes nunca dudaron de mis capacidades y promovieron en mí la confianza de seguir adelante a pesar de las adversidades y los obstáculos que se me han presentado. Les agradezco todos esos esfuerzos, sacrificios, desveladas, paciencia... pero sobre todo, ese amor que ha acompañado a mis hermanos y a mí en nuestras vidas.

A mis hermanos:

- ***Nely***, por mostrarme la forma divertida de la vida, por darme aliento y apoyo, pero sobre todo por enseñarme a sonreír.
- ***Christian***, a pesar de que no haber estado mucho tiempo conmigo, me mostraste la nobleza y la fuerza del alma. Me enseñaste a gozar la vida sin miedo, como si hoy fuera el último día de mi existencia.
- ***Diana***, porque iluminaste mi vida, dándole sentido cuando más lo necesitaba.

Gracias a esas personitas especiales, mis amigos, quienes desinteresadamente me brindaron su invaluable amistad y apoyo; que han estado a mi lado, tanto en mis éxitos como en mis fracasos, siempre con una sonrisa y una palabra de aliento, ayudándome a continuar en pie.

También quiero expresar mi agradecimiento a la empresa *Integradores de Soluciones Empresariales en Tecnologías de la Información*, pues en ella me he desarrollado profesionalmente y en donde conocí a muchas personas que influyeron en mí, enseñándome una nueva perspectiva de la vida.

Pero gracias a ti, que de manera directa o indirecta, me ayudaste a llegar a la culminación de este proyecto.

**"Las metas son como las estrellas, están siempre fijas. Las adversidades como las nubes, son pasajeras. Fija la mirada en las estrellas."**
**(Anónimo)**

Claudia Jazmín De León Cruz

# *Agradecimientos*

Finalmente, después de un largo tiempo, muchos esfuerzos y anhelos se ven reflejados en esta tesis.

No me refiero sólo a la realización de una investigación y al desarrollo de un sistema con el objetivo personal de terminar un ciclo en mi vida académica y de adquirir los conocimientos que de esa actividad emanan.

Hablo también de las personas y los ideales que a lo largo de los años han contribuido a mi crecimiento y bienestar en todos los aspectos que se puedan abrazar.

En primer lugar quiero mostrar mi gratitud hacia mi familia. Porque su apoyo, comprensión, respeto y valores transmitidos se han convertido en la roca más firme, la montaña más alta y el refugio más cálido, sobre los que puedo construir mis proyectos, apreciar el mundo con una visión amplia y sentirme cobijado.

Agradezco también a mis amigos y compañeros de carrera. Pues su compañía y complicidad me han hecho sentir completo. Gracias por su confianza y por compartir conmigo los trances amargos pero, sobre todo, ¡gracias por nuestras celebraciones y por crear esos buenos momentos!

Deseo brindar un reconocimiento especial a mis profesores, pues más allá de sus valiosas enseñanzas en clase, han sido fuente de admiración y ejemplo a seguir en mi vida profesional.

Un profundo y cariñoso agradecimiento a la Facultad de Ingeniería y a la Universidad Nacional Autónoma de México porque la estancia en sus aulas me ha llenado de satisfacciones en muchos niveles. El formar parte de su comunidad es motivo de un gran orgullo y me siento honrado de contar con ese privilegio.

También quiero agradecer al pueblo de México, porque el esfuerzo de todos los que en él laboran ha financiado sustancialmente mi educación y albergo la esperanza de retribuir la oportunidad que me han brindado.

Espero que este trabajo de tesis represente el cofre donde guardo el cariño, el respaldo y el aprecio que me han mostrado y, al mismo tiempo, sea el manifiesto público, claro y sincero de mi infinita gratitud.

**“Por mi raza hablará el espíritu”**

José de Jesús Navarro Carranza

# *Reconocimientos*

Durante el desarrollo del presente proyecto de tesis recibimos una invitación por parte de la Facultad de Ingeniería, a través de su Departamento de Ingeniería en Computación y con la colaboración de su Sociedad de Alumnos, a participar en las Jornadas de Ingeniería en Computación que se celebraron del 29 de agosto al 2 de septiembre de 2005 y cuyo tema principal fue “Computación de calidad para ingeniería de calidad”.

Por tal motivo se brindó una breve ponencia a la comunidad de la Facultad y al público en general acerca de los fundamentos teóricos de los sistemas de base de datos distribuida y se describió la manera en que se estaba desarrollando este proyecto, así como los objetivos principales a alcanzar. Amablemente, nos fueron entregadas sendas constancias de participación al final del evento.

“Desarrollo de un sistema de base de datos distribuida en un cluster de alto desempeño”. Claudia Jazmín de León Cruz, José de Jesús Navarro Carranza, Ing. Laura Sandoval Montaño. Conferencia basada en los capítulos I, II, III, IV y V de esta tesis. Jornadas de Ingeniería en Computación 2005. Minicongreso de Tesistas. 1 de septiembre de 2005. Facultad de Ingeniería, UNAM.

# Índice

# INTRODUCCIÓN

El advenimiento de la globalización ha creado un clima cada vez más competitivo en el que muchas instituciones deben trabajar de una nueva manera, que maximice la cooperación entre las diferentes entidades que las conforman. La inminente utilización de las redes de computadoras ha propiciado la necesidad de un rápido crecimiento para suministrar acceso a distintas fuentes de datos. Esta integración de datos dispersos en diferentes sitios puede requerir de nuevas arquitecturas y herramientas de software, como es el caso de un sistema de base de datos distribuida.

La evolución de las bases de datos distribuidas se debe por una parte a razones organizacionales de instituciones que requieren la integración de información desde distintos sitios, por ejemplo, para una consulta. A su vez, el desarrollo de las tecnologías de comunicación ha permitido enlazar datos con aplicaciones que se encuentran localizadas en distintos sitios de una red. Las transacciones bancarias realizadas en cajeros automáticos que se encuentran ubicados en centros comerciales, empresas y escuelas, no serían posibles si no existieran sistemas de comunicación para enlazar diferentes bases de datos.

La necesidad de interconexión entre bases de datos existentes surge ante el planteamiento de un modelo poco óptimo en donde aplicaciones independientes acceden a sus propias fuentes de información de manera aislada. Este enfoque trae mejores beneficios de seguridad y disponibilidad de la información, ya que la implantación de los mecanismos de control de acceso es más sencilla. Sin embargo cuando existe la necesidad de transferir datos entre diferentes sistemas hay problemas de consistencia y rendimiento. Las bases de datos distribuidas son la solución natural cuando existen diversas bases de datos y se tiene la necesidad de ejecutar aplicaciones globales. En este caso, la base de datos distribuida es creada por encima de las bases de datos locales preexistentes. La descentralización involucra en varios casos el manejo de sistemas heterogéneos. La heterogeneidad se puede dar a muchos niveles, desde la forma de concebir los datos hasta los medios de almacenamiento para mantener su durabilidad, pasando por diferentes sistemas operativos o sistemas de comunicación para transportar la información.

Existen organizaciones que crecen adicionando nuevas unidades relativamente autónomas como sucursales, nuevos almacenes o fábricas, lo que implica el empleo de nuevas bases de datos para los sistemas de información que se desarrollen. Es aquí donde un enfoque de bases de datos

distribuida es muy útil, ya que éstas soportan un suave crecimiento con un mínimo impacto en las unidades con las que ya se cuenta. Además, la existencia de diversos procesadores autónomos resulta en un incremento del rendimiento a través de un alto grado de paralelismo.

Las redes de comunicación son un punto fundamental para que las bases de datos pasen de un escenario centralizado a uno distribuido. El uso de bases de datos distribuidas permite el escalamiento de los recursos de cómputo en forma paulatina sin tener que adquirir un sistema nuevo completo. Sin embargo, aún existen áreas en las bases de datos distribuidas que se encuentran en investigación y desarrollo, las cuales son un reto tecnológico para varios grupos de investigadores.

La creación de una base de datos a la que puedan acudir los usuarios para hacer consultas y acceder a la información que les interese es, pues, una herramienta imprescindible en cualquier sistema informático, desatacando la utilización de bases de datos distribuidas pues cada vez es más corriente el uso de arquitecturas cliente-servidor y trabajo en red. Sin embargo, los sistemas distribuidos de bases de datos constituyen un tema nuevo y por lo cual no muy utilizado, pero con grandes beneficios si se logra explotar al máximo. En ello radica el motivo y la importancia de esta tesis: aplicar los conocimientos adquiridos durante la carrera sobre las bases de datos para construir un sistema manejador de base de datos distribuida y, con ello, facilitar la comprensión de esta tecnología.

El proyecto involucra un trabajo de investigación acerca de los aspectos esenciales de las bases de datos distribuidas, así como el desarrollo de un sistema que permita apreciar de manera práctica el procesamiento distribuido de diversas fuentes de información. El objetivo principal es establecer los fundamentos teóricos de las bases de datos distribuidas, brindar los antecedentes mínimos necesarios para su comprensión, ejemplificar de manera práctica la distribución de información en un *cluster* tipo *Beowulf*, utilizando software libre y tomando en cuenta los recursos disponibles en dicho desarrollo, así como medir el desempeño del procesamiento distribuido de bases de datos en comparación con una base de datos centralizada.

En el primer capítulo, *El cómputo paralelo y distribuido*, se brinda un panorama general de las diversas arquitecturas de computadoras existentes, secuenciales y paralelas, para definir un sistema conocido como *cluster*, que es el tipo de arquitectura paralela en el que se desarrolla el proyecto. Se definen también los sistemas distribuidos en general y se mencionan sus principales características.

El segundo capítulo, *Sistemas De Base De Datos Multiusuarios*, muestra la evolución que han presentado los sistemas de base de datos que proveen servicios a diferentes usuarios, desde los sistemas de teleprocesamiento hasta los sistemas cliente-servidor, para introducir finalmente a los sistemas de base de datos distribuida.

El tercer capítulo, *Bases De Datos Distribuidas*, provee los fundamentos teóricos de este tipo de sistemas de información. Se tratan sus características más importantes, las diferentes maneras en que pueden modelarse y construirse, abarcando aspectos como fragmentación y replicación de una base de datos. También se mencionan sus ventajas, desventajas y diferencias en comparación con un sistema centralizado.

El cuarto capítulo, *El Cómputo Distribuido Con Java RMI*, explica de manera teórica el funcionamiento del sistema de objetos distribuidos del lenguaje Java, utilizado para el desarrollo del proyecto de tesis, se describen sus principales componentes y se brinda un ejemplo práctico de su implementación.

El quinto capítulo, *Transacciones Distribuidas*, involucra los aspectos esenciales del manejo de transacciones en un ambiente distribuido. Se hace hincapié en la importancia de un manejador de transacciones distribuidas y se explica el protocolo *commit* de dos fases con el que dicho manejador consigue este tipo de transacciones.

El sexto capítulo, *Desarrollo de un Sistema de Base de Datos Distribuida*, describe la instalación del software utilizado en el proyecto de tesis y explica a nivel de código la interacción entre las diferentes clases, interfaces y objetos que componen el sistema de base de datos distribuida desarrollado.

Finalmente, se mencionan las características más sobresalientes del sistema y se muestran los resultados obtenidos en el procesamiento distribuido de una base de datos, haciendo una comparación contra los resultados obtenidos en el procesamiento de una base de datos centralizada y, a manera de apéndice, se presenta un apartado con el código del sistema.

# CAPÍTULO 1
# El cómputo paralelo y distribuido

Durante varios años el modo habitual de computación ha sido el secuencial. Así, una computadora con un solo procesador sólo cuenta con la capacidad de ejecutar una instrucción en un momento determinado. Esto se debe a la adopción del modelo propuesto por Von Neuman, cuya estabilidad ha sido la base para el desarrollo de diversas aplicaciones. Sin embargo, desde los primeros días de la computación secuencial se vislumbraba que, tarde o temprano, se tendría que recurrir a la computación paralela para resolver problemas que demandaban mucho poder de cómputo por lo que su resolución en máquinas secuenciales no sería óptima o posible.

## 1. 1. El Cómputo Paralelo

Se le llama cómputo paralelo a la ejecución de dos o más procesos al mismo tiempo usando más de un procesador. El procesamiento en paralelo hace que un programa se ejecute rápidamente porque existen más procesadores que trabajan en forma simultánea y coordinada en diferentes partes de un problema. La meta es reducir al mínimo el tiempo total de cómputo distribuyendo la carga de trabajo entre los procesadores disponibles.

Las aplicaciones de computación paralela se pueden encontrar en una amplia gama de disciplinas, tales como la predicción del clima, el modelado de la biosfera, la exploración petrolera, el procesamiento de imágenes, la fusión nuclear, el modelado de océanos, y muchas otras.

La velocidad de procesamiento no es la única razón para utilizar el paralelismo. La construcción de aplicaciones más complejas ha requerido computadoras más rápidas, y las limitaciones en el desarrollo de computadoras seriales han llegado a ser más y más evidentes.

Anteriormente, el procesamiento paralelo se empleaba para resolver problemas que requerían una gran escala simulación (por ejemplo, de simulación molecular, simulación de una explosión nuclear), un gran número de cálculos y procesamiento de datos (por ejemplo, el cómputo de los datos de un censo).

En la actualidad, el costo del hardware disminuye rápidamente, por lo que el procesamiento paralelo se está empleando cada vez más en tareas comunes. No obstante, además de las facilidades de hardware que permiten llevar a cabo este tipo de procesamiento, es necesario contar con software que soporte la ejecución y la coordinación de procesos en forma paralela.

### 1. 1. 1. Clasificación De Las Arquitecturas De Computadoras

En 1966 Michael Flynn propuso un mecanismo de clasificación de las computadoras. La taxonomía de Flynn es la manera clásica de organizar las computadoras y, aunque no cubre todas las posibles arquitecturas, facilita su comprensión de manera significativa. El método de Flynn se basa en el número de instrucciones y en la secuencia de datos que utiliza una computadora para procesar información. Puede haber secuencias de instrucciones sencillas o múltiples y secuencias de datos sencillas o múltiples. Esto da lugar a cuatro tipos de computadoras, de las cuales solamente dos son consideradas como de procesamiento paralelo.

**Modelo SISD (*Single instruction* - *Single data*)**

Este es el modelo tradicional de computación secuencial donde una unidad de procesamiento recibe una sola secuencia de instrucciones que opera en una secuencia de datos.


Figura 1.1. Modelo SISD

Ejemplo. Para procesar la suma de N números $a_1$, $a_2$, ... $a_N$ , el procesador necesita acceder a memoria N veces consecutivas (para recibir un número). También son ejecutadas en secuencia N-1 adiciones. Los algoritmos para las computadoras SISD no contienen ningún paralelismo ya que están constituidas por un procesador.

### Modelo SIMD (*Single instruction - Multiple data*)

A diferencia de las computadoras SISD, en este caso se tienen múltiples procesadores que sincronizadamente ejecutan la misma secuencia de instrucciones, pero en diferentes datos. El tipo de memoria que estos sistemas utilizan es distribuida, como se verá más adelante.

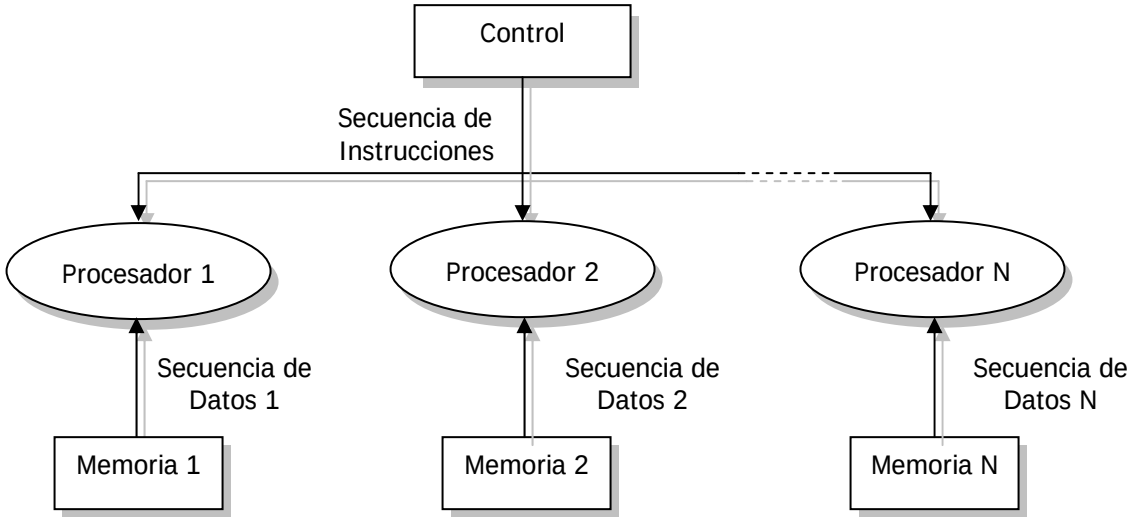



Figura 1.2. Modelo SIMD

Aquí hay N secuencias de datos, una por procesador, así que diferentes datos pueden ser empleados en cada uno de ellos. Los procesadores operan sincronizadamente y un reloj global se utiliza para asegurar esta operación. En cada paso todos los procesadores ejecutan la misma instrucción, cada uno en diferentes datos.

Ejemplo. Para sumar dos matrices cuadradas A y B, cada una de orden 2, con 4 procesadores:

$$A=\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \qquad B=\begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \qquad C=\begin{bmatrix} C_{11}=A_{11}+B_{11} & C_{12}=A_{12}+B_{12} \\ C_{21}=A_{21}+B_{21} & C_{22}=A_{22}+B_{22} \end{bmatrix}$$

| $C_{11} = A_{11} + B_{11}$ | $C_{12} = A_{12} + B_{12}$ | $C_{21} = A_{21} + B_{21}$ | $C_{22} = A_{22} + B_{22}$ |
|---|---|---|---|
| Procesador 1 | Procesador 2 | Procesador 3 | Procesador 4 |

La misma instrucción es ejecutada en los 4 procesadores (suma de dos números) y los 4 ejecutan las instrucciones simultáneamente sobre datos diferentes. Esto toma un paso en comparación con cuatro pasos en una máquina secuencial.

### Modelo MISD (Multiple Instruction - *Single data*)

En este modelo diferentes secuencias de instrucciones pasan a través de múltiples procesadores, cada uno con su propia unidad de control, pero compartiendo una memoria común.

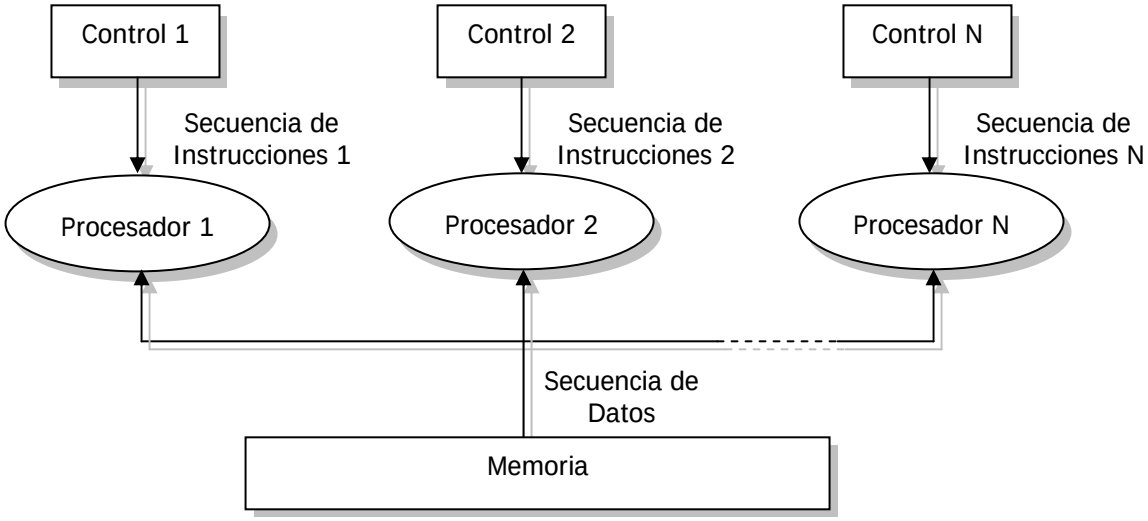



Figura 1.3. Modelo MISD

Aquí hay N secuencias de instrucciones y una secuencia de datos. El paralelismo es alcanzado dejando que los procesadores realicen diferentes operaciones al mismo tiempo sobre datos idénticos. Las máquinas MISD son útiles en cómputos donde una sola entrada está sujeta a diferentes operaciones.

Ejemplo. Para sumar, restar, multiplicar y dividir dos números *a* y *b* con cuatro procesadores:

| S = a+b | R = a-b | M = a*b | D = a/b |
|---|---|---|---|
| Procesador 1 | Procesador 2 | Procesador 3 | Procesador 4 |

Cada uno de los procesadores opera sobre la misma secuencia de datos (a y b), pero en cada caso se tiene una secuencia de instrucciones diferente (suma, resta, multiplicación y división).

### Modelo MIMD (Multiple Instruction - *Multiple data*)

Este tipo de computadora es paralela al igual que las SIMD, la diferencia con estos sistemas es que MIMD es asíncrono. No tiene un reloj central. Cada procesador en un sistema MIMD puede ejecutar su propia secuencia de instrucciones y tener sus propios datos. Esta característica es la más general y poderosa de esta clasificación.

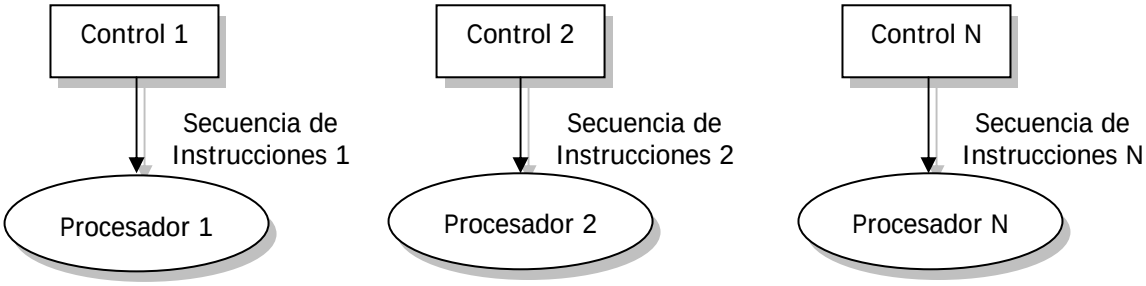



Figura 1.4. Modelo MIMD

Se tienen N procesadores, N secuencias de instrucciones y N secuencias de datos. Cada procesador opera bajo el control de una secuencia de instrucciones, ejecutada por su propia unidad de control, es decir, cada procesador es capaz de ejecutar su propio programa con diferentes datos. Esto significa que los procesadores operan asíncronamente o, en términos simples, pueden realizar diferentes operaciones en diferentes datos al mismo tiempo.

Ejemplo. Si se tienen los números *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h* y se desea calcular la suma de *a* y *b*, la resta de *c* y *d*, la multiplicación de *e* y *f*, y la división de *g* y *h* con cuatro procesadores:

| S = a+b | R = c-d | M = e*f | D = g/h |
|---|---|---|---|
| Procesador 1 | Procesador 2 | Procesador 3 | Procesador 4 |

Cada uno de los procesadores opera sobre una secuencia de datos diferente aplicando una secuencia de instrucciones diferente.

Los sistemas MIMD se clasifican en:

- Sistemas de Memoria Compartida.
- Sistemas de Memoria Distribuida.

### Modelo MIMD de Memoria Compartida

En este tipo de sistemas cada procesador tiene acceso a toda la memoria, es decir, hay un espacio de direccionamiento compartido. Se tienen tiempos de acceso a memoria uniformes ya que todos los procesadores se encuentran igualmente comunicados con la memoria principal y las lecturas y escrituras de todos los procesadores tienen exactamente las mismas latencias; además, el acceso a memoria es por medio de un ducto común. En esta configuración, debe asegurarse que los procesadores no tengan acceso simultáneamente a las mismas regiones de memoria propiciando la ocurrencia de alguna falla.

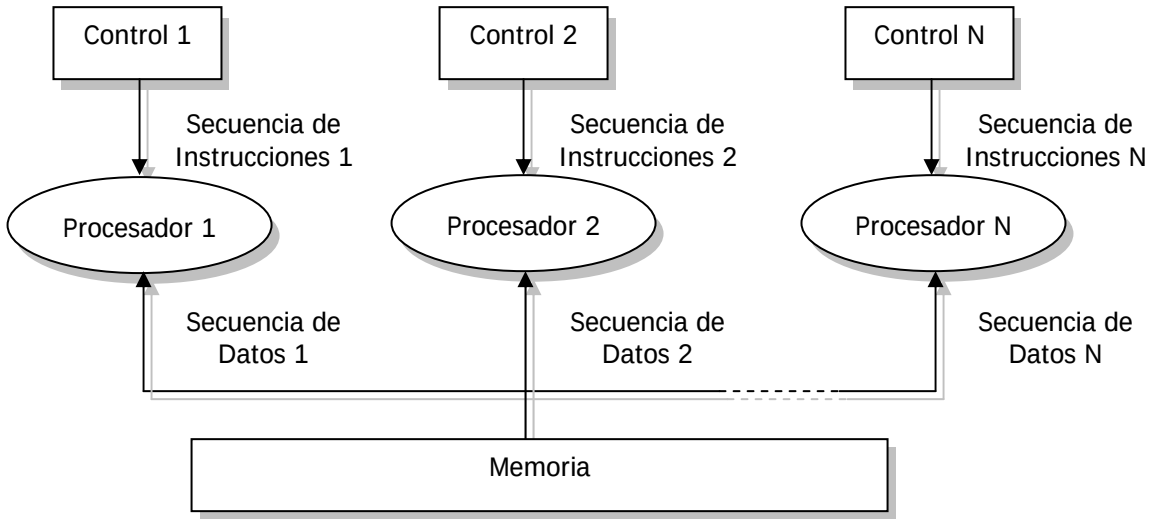



Figura 1.5. Modelo MIMD de Memoria Compartida

Desventajas:

- El acceso simultáneo a memoria es un problema.
- Poca escalabilidad de procesadores, debido a que se puede generar un cuello de botella al incrementar el número de unidades de procesamiento compartiendo el mismo acceso a memoria.

Ventaja:

- La facilidad de la programación. Es mucho más fácil programar en estos sistemas que en sistemas de memoria distribuida.

Las computadoras MIMD con memoria compartida son sistemas conocidos como de multiprocesamiento simétrico (SMP) donde múltiples procesadores comparten un mismo sistema operativo y memoria. Otros términos con los que se les conoce son máquinas fuertemente acopladas o de multiprocesadores.

### Modelo MIMD de Memoria Distribuida

Estos sistemas cuentan con una memoria local propia para cada uno de los procesadores. Éstos pueden compartir información solamente enviando mensajes, es decir, si un procesador requiere los datos contenidos en la memoria de otro procesador, deberá enviar un mensaje solicitándolos. A esta comunicación se le conoce como paso o transferencia de mensajes.

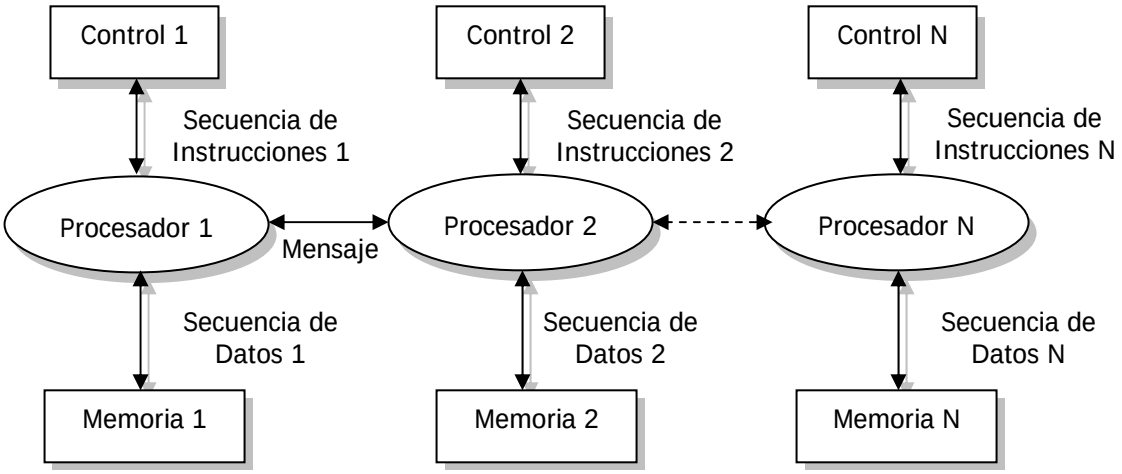



Figura 1.6. Modelo MIMD de Memoria Distribuida

Ventajas:

- La escalabilidad. Las computadoras con sistemas de memoria distribuida son fáciles de escalar. Mientras que la demanda de los recursos crece se puede agregar más memoria y procesadores.

Desventajas:

- El acceso remoto a memoria puede ser lento debido a la transferencia de mensajes.
- La programación es más complicada que en los sistemas de memoria compartida.

Las computadoras MIMD de memoria distribuida son conocidas como sistemas de procesamiento en paralelo masivo o MPP (*Massively Parallel Processing*) donde múltiples procesadores trabajan en diferentes partes de un programa, usando su propio sistema operativo y memoria. Además se les llama multicomputadoras o máquinas débilmente acopladas.

### 1. 1. 2. *Cluster*s

Con la potencia de cálculo y los bajos costos de las computadoras personales actuales, la disponibilidad de equipos conectados a través de una red de comunicaciones, los sistemas operativos de dominio público y la expansión del software libre, es cada vez más factible construir ambientes de cómputo de alto rendimiento en los cuales se pueda obtener un buen desempeño por medio del cómputo paralelo. Un buen ejemplo de este tipo de ambientes es el llamado *cluster*.

Un *cluster* es un caso especial de un sistema MIMD de memoria distribuida. En él, un grupo de computadoras débilmente acopladas trabajan en conjunto y crean la ilusión de ser un único recurso de cómputo.

Básicamente, cualquier conjunto de sistemas interconectados por una red y dedicados a un propósito específico puede ser llamado *cluster*. La diferencia entre un *cluster* y un grupo de computadoras conectadas por una red es que las computadoras en un *cluster* funcionan como una unidad; situación que no se presenta en cualquier red de computadoras.

#### *Beowulf*

Un *cluster* tipo *Beowulf* es aquél en el cual las computadoras que lo componen son fáciles de conseguir, no contienen algún componente de hardware específico y, usualmente, utilizan el sistema operativo Linux.

Existe un gran número de opciones al planear un nuevo *cluster* tipo *Beowulf*. Las configuraciones de este tipo de *cluster* difieren entre sí en cuanto a seguridad, software de aplicación, administración, sistema de arranque y de archivos, componentes de red y otras características, aunque son similares en su naturaleza en términos de propósito y concepto.

Algunos *cluster*s *Beowulf* consisten de máquinas sin disco duro mientras que en otros cada una puede contar con sus propios dispositivos de almacenamiento, sistema operativo y sistema de archivos. Algunos pueden tener un *gateway* como enlace con el mundo exterior, pero en otros cada nodo puede tener su propia dirección IP y ser accesible desde fuera del *cluster*.

Algunos aspectos que se tienen en cuenta al establecer un *cluster* tipo *Beowulf* son:

- Un *cluster* puede construirse con base en un grupo de computadoras personales, las cuales pueden lograr un desempeño tan bueno como las computadoras paralelas comerciales.
- Casi todo el software utilizado en un *cluster* tipo *Beowulf* está disponible bajo la licencia pública GNU o bajo licencias que permiten el acceso a archivos fuente de software libre.
- Un *cluster Beowulf* es fácil de actualizar con una fracción del dinero que se utilizaría sólo para el mantenimiento de una computadora paralela comercial.

Entre las diferentes plataformas de cómputo distribuido que existen actualmente, el costo que implica el uso de software propietario, el mantenimiento de la infraestructura de un sistema paralelo, los recursos humanos y operacionales requeridos, el *cluster* tipo *Beowulf* representa uno de los más económicos. Las máquinas que lo componen no necesitan de personal especializado para su mantenimiento. Si alguna de ellas se avería puede repararse con poco dinero o ser sustituida por una máquina más rápida y actual. Una máquina que no funciona correctamente puede reemplazarse o retirarse del *cluster* sin afectar notablemente su desempeño.

### El *cluster Beowulf* de la Facultad de Ingeniería.

El *cluster Beowulf* en el que se desarrolla el proyecto de tesis cuenta con tres nodos, cuyas características se resumen en la figura 1.7:

| | Nodo de Acceso y Procesamiento | Nodo de Procesamiento | Nodo de Procesamiento |
|---|---|---|---|
| |  |  | |
| Compañía | Dell | Dell | Dell |
| Modelo | PowerEdge 600SC | Optiplex GXpro | Optiplex GXpro |
| Procesador | Intel Pentium 4 | Intel Pentium Pro | Intel Pentium Pro |
| Velocidad | 2.4 GHz | 180 MHz | 180 MHz |
| Sistema Operativo | Red Hat Linux 8.0 Modo Gráfico y Consola | Red Hat Linux 7.2 Modo Consola | Red Hat Linux 7.2 Modo Consola |
| Memoria RAM | 1 GB | 512 MB | 512 MB |
| Tarjeta de Red | Intel Kenai 32 Gigabit para red Ethernet a 10/100/1000 Mbps con conector RJ-45 | 3Com 3C590 para red Ethernet a 10 Mbps con conector RJ-45 | 3Com 3C590 para red Ethernet a 10 Mbps con conector RJ-45 |
| Cable de Red | Cable UTP Categoría 5e con conectores RJ-45 | | |

Figura 1.7. Características de los nodos del *cluster beowulf*.

El único nodo que cuenta con periféricos tales como teclado y monitor es el nodo de acceso, a través del cual los usuarios del *cluster* hacen uso del sistema. Los nodos restantes sólo son utilizados para ayudar en la ejecución de procesos.

En la tabla anterior se puede apreciar que el nodo de acceso al *cluster* cuenta con una potencia de procesamiento mayor, lo que se debe tomar en cuenta al analizar las pruebas de desempeño realizadas con el sistema desarrollado en el presente trabajo de investigación.

### 1. 2. El Cómputo Distribuido

Un sistema distribuido se define como una colección de computadoras independientes o autónomas interrelacionadas a través de una red y que cuenta con software diseñado para proveer servicios de cómputo, integrados de tal forma que ante sus usuarios aparece como un único sistema.

El desarrollo de los sistemas distribuidos vino de la mano de las redes locales de alta velocidad a principios de 1970. El concepto abstracto más utilizado en el campo de cómputo distribuido son las llamadas a procedimientos remotos o RPC (*Remote Procedure calls*). Los RPC permiten a una función remota ser llamada como si se tratara de una local.

Los sistemas distribuidos se implementan en diversas plataformas hardware, desde unas pocas estaciones de trabajo conectadas por una red de área local hasta Internet: una colección de redes de área local y de área extensa interconectadas, que enlazan millones de computadoras.

Las aplicaciones de los sistemas distribuidos varían desde la provisión de capacidad de cómputo a grupos de usuarios, hasta sistemas bancarios, comunicaciones multimedia y abarcan prácticamente todas las aplicaciones comerciales y técnicas de las computadoras.

#### 1. 2. 1. Características de los sistemas distribuidos

- **Compartimiento de Recursos**

El término recurso es bastante abstracto, pero es el que mejor caracteriza el abanico de entidades que pueden compartirse en un sistema distribuido. El abanico se extiende desde dispositivos electrónicos, como discos e impresoras, hasta elementos software como archivos, bases de datos y otros componentes.

Los usuarios de computadoras personales dentro de un sistema distribuido no obtienen automáticamente los beneficios del compartimiento de recursos. Los recursos en un sistema distribuido están físicamente encapsulados en una de las computadoras y sólo pueden ser accedidos por otras computadoras mediante una red de comunicaciones. Para que el compartimiento de recursos sea efectivo, debe ser manejado por un programa que ofrezca una interfaz de comunicación para que el recurso sea accedido, manipulado y actualizado de una manera fiable y consistente.

Un manejador de recursos es un módulo de software que maneja un conjunto de recursos de un tipo en particular. Cada tipo de recurso requiere algunas políticas y métodos específicos junto con requisitos comunes para todos ellos. Tales manejadores incluyen la provisión de un esquema de nombres para cada clase de recurso, el acceso a recursos individuales desde cualquier localización, la traslación del nombre de un recurso a direcciones de comunicación y la coordinación de accesos concurrentes que cambian el estado de los recursos compartidos para mantener la consistencia.

Un sistema distribuido puede verse de manera abstracta como un conjunto de manejadores de recursos y un conjunto de programas que los utilizan. Las aplicaciones que brindan servicio a los usuarios se comunican con dichos manejadores para acceder a los recursos compartidos del sistema.

- **Concurrencia**

Cuando existen varios procesos en una única máquina decimos que se están ejecutando concurrentemente. Si la computadora está equipada con un único procesador central, la concurrencia tiene lugar alternando la ejecución de los distintos procesos.

En los sistemas distribuidos hay muchas computadoras, cada una con uno o más procesadores centrales. Si hay N computadoras en un sistema distribuido con un procesador central cada una entonces hasta N procesos pueden estar ejecutándose en paralelo.

- **Escalabilidad**

Los sistemas distribuidos operan de manera efectiva a muchas escalas diferentes. La escala más pequeña consiste en dos nodos y un servidor, mientras que un sistema distribuido construido alrededor de una red de área local simple podría contener varios cientos de nodos, varios

servidores de archivos, servidores de bases de datos, servidores de impresión u otros servidores de propósito especifico.

Tanto el software de sistema como el de aplicación no deben cambiar cuando la escala del sistema se incrementa. La necesidad de escalabilidad no es sólo un problema de prestaciones de red o de hardware, sino que está íntimamente ligada con todos los aspectos del diseño de los sistemas distribuidos. El diseño del sistema debe reconocer explícitamente la necesidad de escalabilidad.

La demanda de escalabilidad en los sistemas distribuidos ha conducido a una filosofía de diseño en que cualquier recurso simple, hardware o software, puede extenderse para proporcionar servicio a tantos usuarios como se requiera. Si la demanda de un recurso crece, debe ser posible extender el sistema para dar servicio.

Cuando el tamaño y la complejidad de las redes de computadoras aumenta, es un objetivo primordial diseñar software de sistema distribuido que siga siendo eficiente y útil con esas nuevas configuraciones de red. El trabajo necesario para procesar una petición de acceso a un recurso debe ser prácticamente independiente del tamaño de la red.

- **Tolerancia a Fallas**

Cuando existen fallas en el software o en el hardware que constituye un sistema informático, los programas pueden producir resultados incorrectos o pueden parar antes de terminar el cómputo que estaban realizando. El diseño de sistemas tolerantes a fallas se basa en dos cuestiones, complementarias entre sí: redundancia de hardware (uso de componentes redundantes) y recuperación del software (diseño de programas que sean capaces de recuperarse de las fallas).

- **Transparencia**

La transparencia se define como la ocultación al usuario de la separación de los componentes de un sistema distribuido, de manera que el sistema se percibe como un todo en vez de una colección de componentes independientes.

Algunas formas de transparencia son:

- **Transparencia de acceso**: Permite el acceso a los recursos remotos de la misma forma que se accede a los recursos locales.

- **Transparencia de localización**: Permite el acceso a los recursos sin conocimiento de su localización.

- **Transparencia de concurrencia**: Permite que varios procesos operen concurrentemente utilizando recursos compartidos de forma que no exista interferencia entre ellos.

- **Transparencia de replicación**: Permite utilizar múltiples instancias de los recursos para incrementar la fiabilidad y las prestaciones sin que los usuarios o los programas de aplicación tengan que conocer la existencia de las réplicas.

- **Transparencia de fallas**: Permite a los usuarios y programas de aplicación completar sus tareas a pesar de la ocurrencia de fallas en el hardware o en el software del sistema.

- **Transparencia de migración**: Permite el cambio de recursos dentro de un sistema sin afectar a los usuarios o a los programas de aplicación.

- **Transparencia de escalado**: Permite la expansión del sistema y de las aplicaciones sin cambiar la estructura del sistema o los algoritmos de la aplicación.

Las dos más importantes son las transparencias de acceso y de localización; su presencia o ausencia afecta fuertemente la utilización de los recursos distribuidos. A menudo se les denomina transparencias de red. La transparencia de red provee un grado similar de anonimato en el acceso a los recursos que el encontrado en los sistemas centralizados.

# CAPÍTULO 2

# Sistemas De Base De Datos Multiusuarios

Un sistema multiusuario es un sistema informático que da servicio, de manera concurrente, a diferentes usuarios mediante la utilización compartida de sus recursos. Con el fin de llevar a cabo su trabajo, cada usuario inicia una sesión por medio de una terminal conectada directamente al sistema o de un cliente ubicado en un sistema remoto conectado por medio de una red de comunicaciones.

Los sistemas de base de datos multiusuarios están soportados por diversas arquitecturas. En el pasado, los más comunes eran los sistemas de teleprocesamiento. Conforme se ha ido reduciendo el precio de las CPU, se ha hecho factible la utilización de más de una computadora, lo cual ha producido nuevas alternativas de bases de datos multiusuarios.

## 2. 1. Sistemas De Teleprocesamiento

El método clásico de soportar un sistema de base de datos multiusuario es el teleprocesamiento, que utiliza una computadora y una unidad central. Todo el procesamiento es efectuado por esta computadora.

En la figura 2.1 se muestra un sistema de teleprocesamiento típico. Los usuarios operan terminales no inteligentes que transmiten a la macrocomputadora mensajes de transacciones y datos. La unidad de teleprocesamiento recibe los mensajes y los datos y los envía al programa de aplicación apropiado. El programa llama al sistema manejador de base de datos (SMBD) solicitando servicios, y éste procesa la base de datos. Terminada la transacción, los resultados son devueltos a los usuarios en las terminales no inteligentes a través de la red.

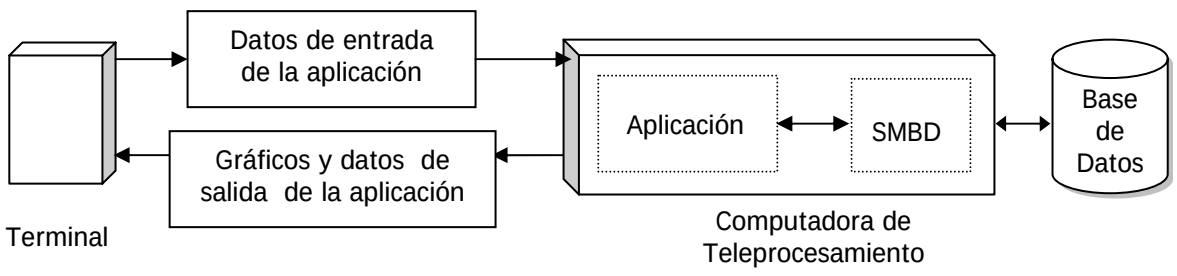



Figura 2.1. Funcionamiento de la arquitectura de teleprocesamiento.

La figura 2.2 muestra N usuarios sometiendo transacciones procesadas por tres distintos programas de aplicación. Dado que existe poca inteligencia en el extremo del usuario (las terminales no son inteligentes), todos los comandos encargados de la interfaz gráfica deben ser generados por la CPU y transmitidos por las líneas de comunicación. Por lo general, la interfaz de usuario está orientada a caracteres y es primitiva. Todas las entradas y las salidas son comunicadas a la macrocomputadora para su procesamiento a distancia.

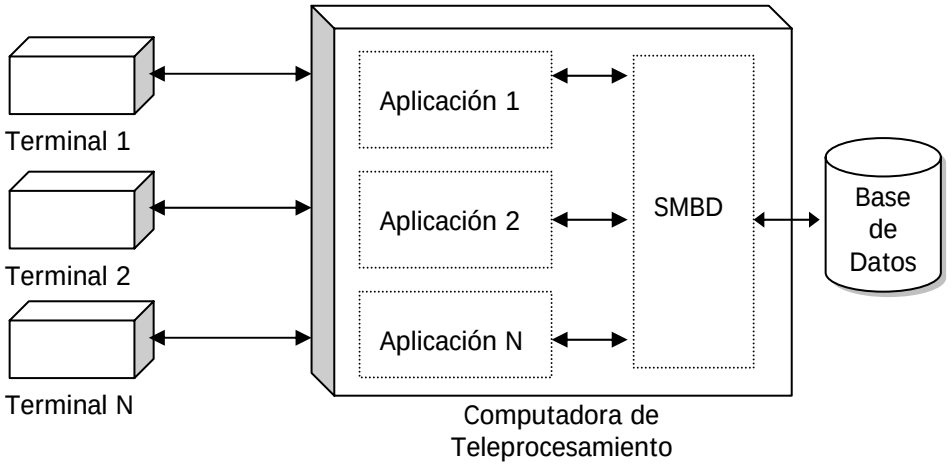



Figura 2.2. Ejemplo de un sistema de teleprocesamiento.

Los sistemas de teleprocesamiento fueron la alternativa más común para el desarrollo de sistemas de base de datos multiusuario. Conforme se ha ido reduciendo la afinidad precio-rendimiento de las computadoras, y con el advenimiento de las microcomputadoras, se han utilizado otras alternativas que requieren de varias computadoras.

## 2. 2. Sistemas Cliente-Servidor

En la figura 2.3 aparece el esquema de una de esas alternativas, llamado sistema cliente-servidor. A diferencia del teleprocesamiento, que implica el uso de una sola computadora, la computación cliente-servidor involucra varias computadoras conectadas por una red. Algunas de las computadoras procesan programas de aplicación y se conocen como clientes. Otra computadora procesa la base de datos y es designada como servidor.

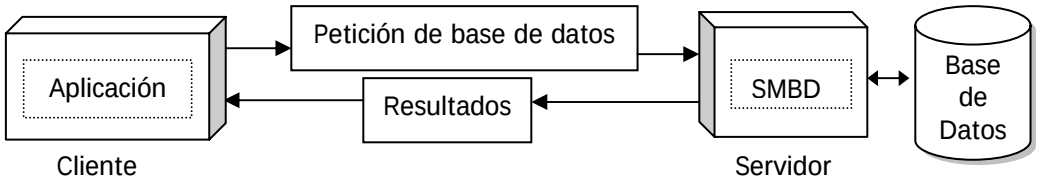



Figura 2.3. Funcionamiento de la arquitectura Cliente-Servidor.

En la figura 2.4 cada uno de los N usuarios tiene su propia computadora de procesamiento de aplicaciones. El Usuario 1 procesa la Aplicación 1 y la Aplicación 2 en la computadora 1. El Usuario 2 procesa Aplicación 2 en la computadora 2, y el Usuario N procesa la Aplicación 2 y la Aplicación 3 en la computadora N. Otra computadora es utilizada como servidor de base de datos.

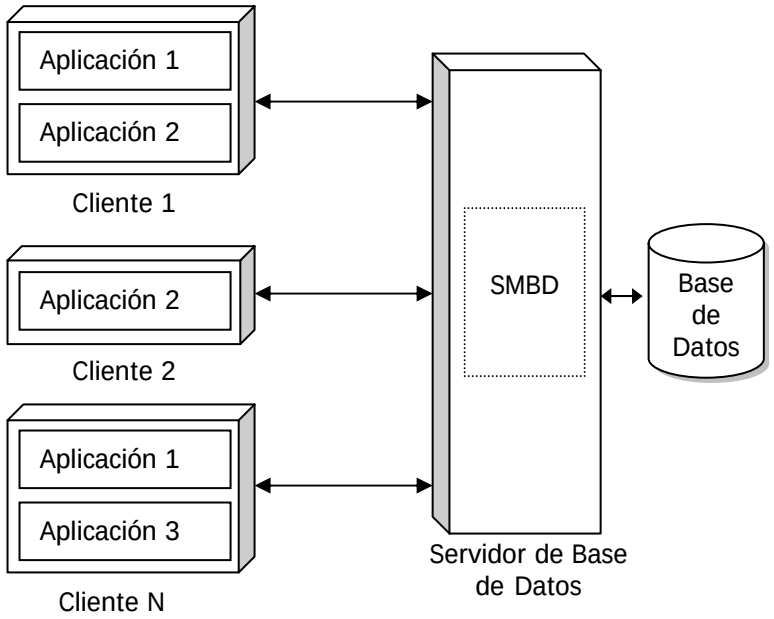



Figura 2.4. Arquitectura cliente servidor

Existen múltiples opciones en relación con el tipo de computadora. En teoría, los clientes y el servidor pueden ser macro, mini, o microcomputadoras. Sin embargo, debido al costo, casi en todos los casos son utilizadas las microcomputadoras. Los clientes y el servidor están conectados, generalmente, por medio de una red de área local.

Existen situaciones en las que el servidor es una macrocomputadora, sobre todo cuando se requiere de considerable potencia de procesamiento o, por razones de tipo político y

organizacional, no resulta apropiado colocar la base de datos en una microcomputadora. Respecto a los clientes, es poco común que éstos no sean microcomputadoras.

El sistema de la figura 2.4. sólo tiene un servidor aunque éste no es siempre el caso. Puede haber múltiples servidores procesando diferentes bases de datos o que proveen otros servicios en beneficio de los clientes. Por ejemplo, en una firma de consultores de ingeniería, un servidor puede procesar la base de datos mientras otro provee apoyo en gráficos asistidos por computadora.

El cliente maneja la interfaz con el usuario. Acepta datos de éste, procesa la lógica de la aplicación y genera peticiones de servicios de base de datos. Los clientes transmiten dichas peticiones al servidor y reciben los resultados, los cuales son transmitidos al usuario de manera adecuada.

El servidor acepta las peticiones de los clientes, las procesa y devuelve un resultado. Mientras hace eso, verifica la integridad de la base de datos y provee control de concurrencia. También se encarga de la recuperación de fallas y optimiza el procesamiento de búsquedas.

En la figura 2.5 se resumen las tareas de un cliente y un servidor:

| **Tareas del CLIENTE** | **Tareas del SERVIDOR** |
|---|---|
| Manejar la interfaz con el usuario<br>Aceptar datos del usuario<br>Procesar la lógica de la aplicación<br>Generar peticiones de base de datos<br>Transmitir peticiones al servidor<br>Recibir resultados del servidor<br>Dar formato a los resultados | Aceptar peticiones de los clientes<br>Procesar peticiones de base de datos<br>Dar formato a los resultados<br>y transmitirlos al cliente<br>Verificar la integridad de la base de datos<br>Proveer el control de concurrencia<br>Realizar la recuperación<br>Optimizar las búsquedas |

Figura 2.5. Tareas de un cliente y un servidor.

Un sistema cliente-servidor sitúa el procesamiento de la aplicación cerca del usuario lo cual trae como consecuencia un incremento en la eficiencia. Diversas computadoras procesan aplicaciones en paralelo. Además, los costos de comunicación se reducen. Sólo las peticiones para el sistema manejador de base de datos y las respuestas a dichas peticiones son enviadas a través de la red. Esto implica un menor tráfico que el que existe en el teleprocesamiento. Dado que

múltiples computadoras procesan aplicaciones y sólo necesitan al servidor para procesar la base de datos, la interfaz con el usuario puede ser más elaborada.

Una desventaja de los sistemas cliente-servidor es el control. Los clientes operan simultáneamente y procesan aplicaciones en paralelo. Esto introduce la posibilidad de pérdidas en actualizaciones y otros problemas de control multiusuario. Dichos problemas son más severos que en los sistemas de teleprocesamiento porque las computadoras realizan solicitudes de manera concurrente. En el teleprocesamiento toda la actividad es gobernada por un único sistema operativo local.

## 2. 3. Sistemas De Recursos Compartidos

Esta arquitectura no sólo distribuye los programas de aplicación sino también la salida del sistema manejador de base de datos a las computadoras procesadoras. En este caso, el servidor es un servidor de archivos y no un servidor de base de datos. Si comparamos las arquitecturas vistas hasta el momento nos daremos cuenta de que cada vez hay más software del lado del usuario.

La arquitectura de recursos compartidos fue desarrollada antes de la arquitectura cliente-servidor y, en muchas formas, es más primitiva que ésta. En un sistema de recursos compartidos, el sistema manejador de base de datos (SMBD) en cada cliente manda una petición de archivos al servidor. Esto implica que un mayor tráfico cruza la red que en una arquitectura cliente-servidor.

Para comprender esto, consideremos el procesamiento de una búsqueda para obtener el nombre (Nombre) y la dirección (Direccion) de todos los registros en la tabla COMPRADOR donde el código postal (CP) sea 98033. En un sistema cliente-servidor, el programa de aplicación enviaría el siguiente comando SQL:

```
SELECT Nombre, Direccion
FROM COMPRADOR
WHERE CP = 98033
```

El servidor respondería con todos los nombres y direcciones requisitados, como se puede apreciar en la figura 2. 6.

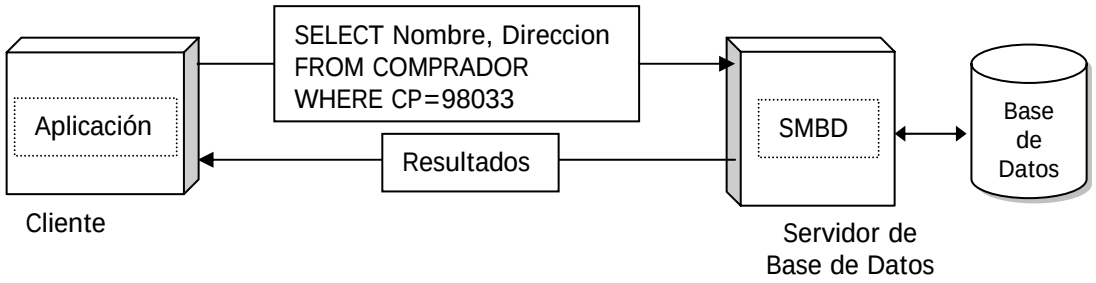



Figura 2.6. Procesamiento de una búsqueda en un sistema cliente-servidor.

En un sistema de recursos compartidos, el SMBD se encuentra en la computadora local. Por lo tanto, ningún programa en el servidor de archivos es capaz de procesar un comando SQL o de algún lenguaje similar. Tal tipo de procesamiento debe hacerse en la computadora del usuario. Así, el SMBD debe mandar una petición al servidor de archivos para que éste transmita la tabla COMPRADOR completa. Además, si la tabla tiene índices u otros elementos asociados, las estructuras asociadas también deben enviarse.

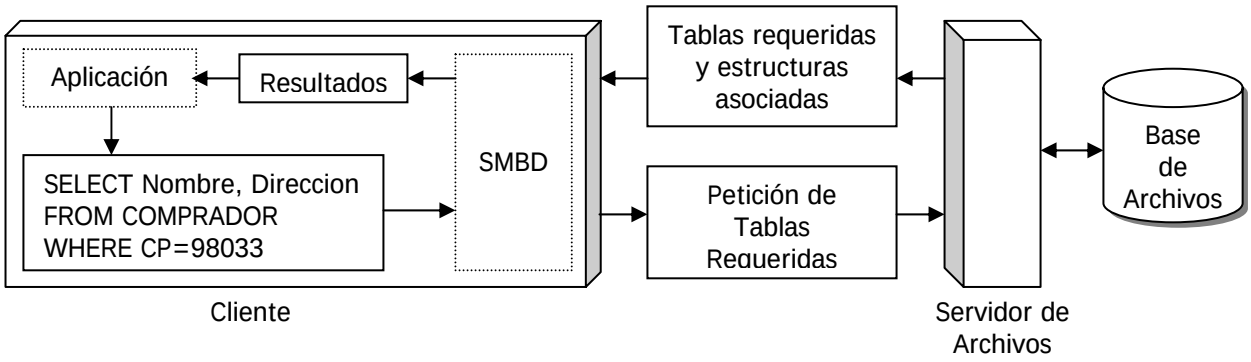



Figura 2.7. Procesamiento de una búsqueda en un sistema de recursos compartidos.

Como se puede ver en la figura 2.7 es necesario transmitir muchos más datos a través de la red. Por si esto fuera poco, mientras un usuario está procesando una petición, grandes porciones de la base de archivos deben bloquearse. Un alto nivel de granularidad de bloqueo es requerido. Así, la cantidad de respuestas del sistema en un determinado periodo se vería reducida.

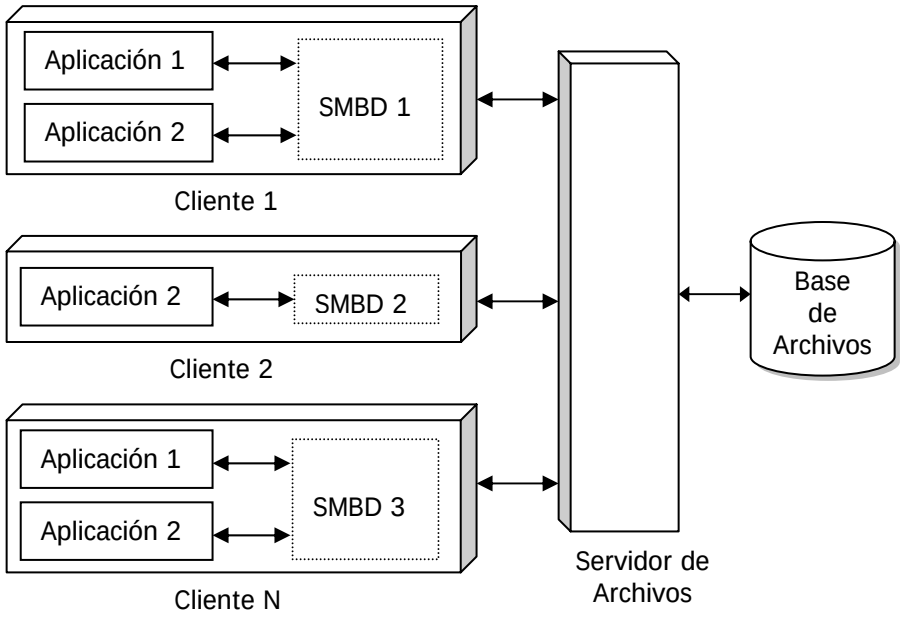



Figura 2.8. Arquitectura de Recursos Compartidos

Debido a estos problemas, los sistemas de recursos compartidos son esporádicamente utilizados para el procesamiento de bases de datos multiusuarios orientadas a transacciones. Para cada transacción se debe bloquear una gran cantidad de datos hasta que éstos son modificados y enviados de nuevo a la base de archivos. Un intento de usar esta arquitectura en el procesamiento de transacciones produciría un sistema con lento desempeño.

Existe, sin embargo, una aplicación de base de datos para la cual esta arquitectura tiene sentido: el procesamiento de consultas en datos descargados o extraídos. Si uno o más usuarios necesitan acceder a grandes porciones de la base de datos para hacer consultas, producir reportes o responder preguntas, la utilización de un servidor que descargue grandes cantidades de datos puede ser la mejor solución. En este caso, los datos extraídos no son actualizados ni devueltos a la base de datos, sino procesados de manera local para brindar los resultados de la consulta a los usuarios.

## 2. 4. Sistemas De Base De Datos Distribuida

En esta arquitectura, la base de datos o una porción de ella está almacenada en N computadoras. En general, las computadoras involucradas procesan tanto aplicaciones como bases de datos. Esto puede apreciarse en la figura 2.9.

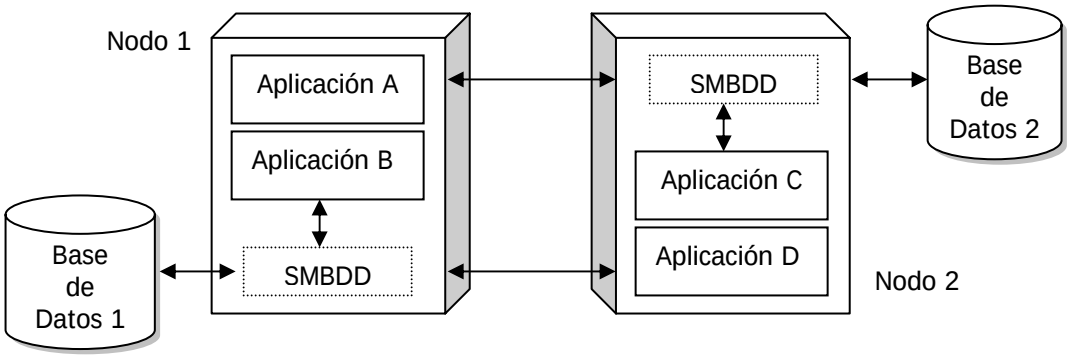



Figura 2.9. Arquitectura de Base de Datos Distribuida.

La base de datos distribuida está compuesta por todas las bases de datos de las N computadoras. Tales computadoras podrían estar localizadas en un mismo lugar, en sitios distintos en el mundo, o algo intermedio.

## 2. 5. El procesamiento distribuido en comparación con el procesamiento distribuido de base de datos.

Las alternativas de recursos compartidos, cliente-servidor, y base de datos distribuida difieren del teleprocesamiento en un aspecto importante: todas ellas utilizan varias computadoras para las aplicaciones o para el procesamiento del SMBD. Por lo tanto, se puede decir que las tres arquitecturas son ejemplos de sistemas distribuidos pues el procesamiento de aplicaciones ha sido distribuido en varias computadoras.

Por otro lado, la base de datos misma queda distribuida en la última arquitectura. Ni la arquitectura cliente-servidor ni la de recursos compartidos distribuyen la base de datos. Ninguna de ellas puede considerarse un sistema de base de datos distribuida.

Cuando se utilizan sistemas de recursos compartidos para procesar datos descargados o extraídos, éstos se localizan en un área intermedia. En vista de que los datos han sido descargados, se trata de una distribución, sin embargo, los datos descargados rara vez se actualizan, por lo que no se distribuyen para todas las funciones.

# CAPÍTULO 3

## Bases de datos distribuidas

La cantidad de innovaciones tecnológicas que se ha dado en las últimas décadas ha promovido cambios en la forma de observar los sistemas de información y, en general, las aplicaciones computacionales. Existen avances tecnológicos que se realizan continuamente en circuitos, dispositivos de almacenamiento, programas y metodologías. Sin embargo, estos cambios tecnológicos van de la mano con la demanda de los usuarios y, por lo tanto, el continuo desarrollo de productos que incorporan nuevas ideas es generado por compañías e instituciones académicas.

Un tema de creciente interés en las esferas tecnológica y académica actuales es el cómputo distribuido. En él, un conjunto de elementos de procesamiento autónomos (no necesariamente homogéneos) se interconectan a través de una red de comunicaciones y cooperan para realizar tareas asignadas.

Entre los términos más comunes que se utilizan para referirse al cómputo distribuido podemos encontrar: funciones distribuidas, procesamiento distribuido de datos, multiprocesadores, multicomputadoras, procesamiento satelital, procesamiento tipo *backend*, computadoras dedicadas y de propósito específico, sistemas de tiempo compartido y sistemas funcionalmente modulares.

Un área en la cual se está integrando la tecnología con nuevas arquitecturas o formas de hacer las cosas es, sin lugar a dudas, el área de los sistemas distribuidos de información. Ellos se refieren al manejo de datos almacenados en equipos de cómputo localizados en muchos sitios, ligados a través de una red de comunicaciones.

Actualmente diversas actividades en las cuales se encuentran involucradas las bases de datos requieren realizarse desde diferentes sitios. Muchas empresas se han diversificado geográficamente y sus recursos de cómputo se encuentran esparcidos. Sin embargo, las consultas de datos entre diferentes unidades de negocios son frecuentes, más aún con el advenimiento de la Internet. Las bases de datos distribuidas, un caso específico de los sistemas de cómputo distribuido, son una buena alternativa para estas situaciones.

La disponibilidad de las bases de datos y de las redes de computadoras ha promovido el desarrollo de este nuevo campo. Una base de datos distribuida es una base de datos integrada y construida por encima de una red de computadoras en lugar de una sola computadora. Las bases de datos distribuidas ofrecen diversas ventajas a los diseñadores y usuarios de bases de datos. Entre las más importantes se encuentra la transparencia en el acceso y en la localización de información. Sin embargo, el diseño y administración de bases de datos distribuidas constituye un gran desafío que incorpora problemas no encontrados en bases de datos centralizadas. Algunos ejemplos son los esquemas de fragmentación y localización de información, el manejo de consultas a sitios distribuidos y los mecanismos de control de concurrencia y confiabilidad.

Los ambientes en los que se encuentra con mayor frecuencia el uso de las bases de datos distribuidas son:

- Hospitales.
- Cadenas hoteleras.
- Líneas de transportación aérea.
- Servicios bancarios y financieros.
- La industria de la manufactura. Particularmente aquella con plantas múltiples.
- Organismos gubernamentales y de servicio público.
- Aplicaciones de control y comando militar.
- Cualquier organización que tiene una estructura descentralizada.

### 3. 1. Conceptos Básicos

Una base de datos distribuida (BDD) es una colección de múltiples bases de datos que están coordinadas y relacionadas de manera lógica y se encuentran distribuidas en una red de computadoras.

A cada una de las computadoras que integran la red se le conoce como nodo o sitio. Cada uno de ellos almacena datos de uso frecuente y los maneja de manera local, pero cuenta con la habilidad de acceder a datos ubicados en otros sitios de la red. Los distintos nodos pueden encontrarse en una habitación, una oficina, una ciudad o a lo largo del planeta. De esta manera, la distancia geográfica no es un factor que determina si una base de datos es distribuida. Una BDD es entonces una colección de datos que pertenecen lógicamente a un solo sistema, pero se encuentra físicamente esparcida en varios sitios de una red.

Un sistema manejador de base de datos distribuida (SMBDD) es aquél que se encarga del manejo de la BDD y proporciona un mecanismo de acceso que hace que la distribución sea transparente a los usuarios.

El término transparente significa que la aplicación trabajaría, desde un punto de vista lógico, como si un solo SMBD, ejecutado en una sola máquina, administrara esos datos.

Un sistema de base de datos distribuida (SBDD) es un caso particular de un sistema de cómputo distribuido. Es un sistema en el cual múltiples sitios de bases de datos están ligados por un sistema de comunicaciones, de tal forma que, un usuario en cualquier sitio puede acceder los datos en cualquier parte de la red exactamente como si los datos estuvieran almacenados en su sitio propio.

De esta manera, en un SBDD se pueden realizar transacciones locales, en las que se accede a datos del nodo donde se inició una operación, y transacciones globales, en las que se accede a datos de nodos distintos a aquél en el que una operación fue iniciada.

Un sistema de base de datos distribuida es entonces el resultado de la integración de una base de datos distribuida (BDD) con un sistema para su manejo (SMBDD):

**SBDD = BDD + SMBDD**

Un sistema de base de datos distribuida se compone de un conjunto de sitios, conectados entre sí mediante algún tipo de red de comunicaciones, en el cual :

- Cada sitio es un sistema de base de datos en sí mismo.
- Los sitios han convenido en trabajar juntos con el fin de que un usuario de cualquier sitio pueda acceder a los datos de cualquier punto de la red tal como si todos los datos estuvieran almacenados en el sitio propio del usuario.

En consecuencia, la llamada base de datos distribuida es en realidad una especie de objeto virtual, cuyas partes componentes se almacenan físicamente en varias bases de datos "reales" distintas, ubicadas en diferentes sitios. De hecho, es la unión lógica de esas bases de datos.

En otras palabras, cada sitio tiene sus propias bases de datos "reales" locales, su propio SMBD y sus programas para el manejo de transacciones. En particular un usuario dado puede realizar operaciones sobre los datos en su propio sitio local exactamente como si ese sitio no participara en absoluto en el sistema distribuido. Así pues, el SBDD puede considerarse como una especie de sociedad entre los SMBD individuales locales de todos los sitios. Un nuevo componente de software en cada sitio, llamado manejador de transacciones distribuidas (MTD), realiza las funciones de sociedad necesarias; y es la combinación de este nuevo componente y los SMBD ya existentes lo que constituye el llamado SMBDD.

Veamos ahora un ejemplo. Considere un banco que tiene tres sucursales, como se muestra en la figura 3.1.

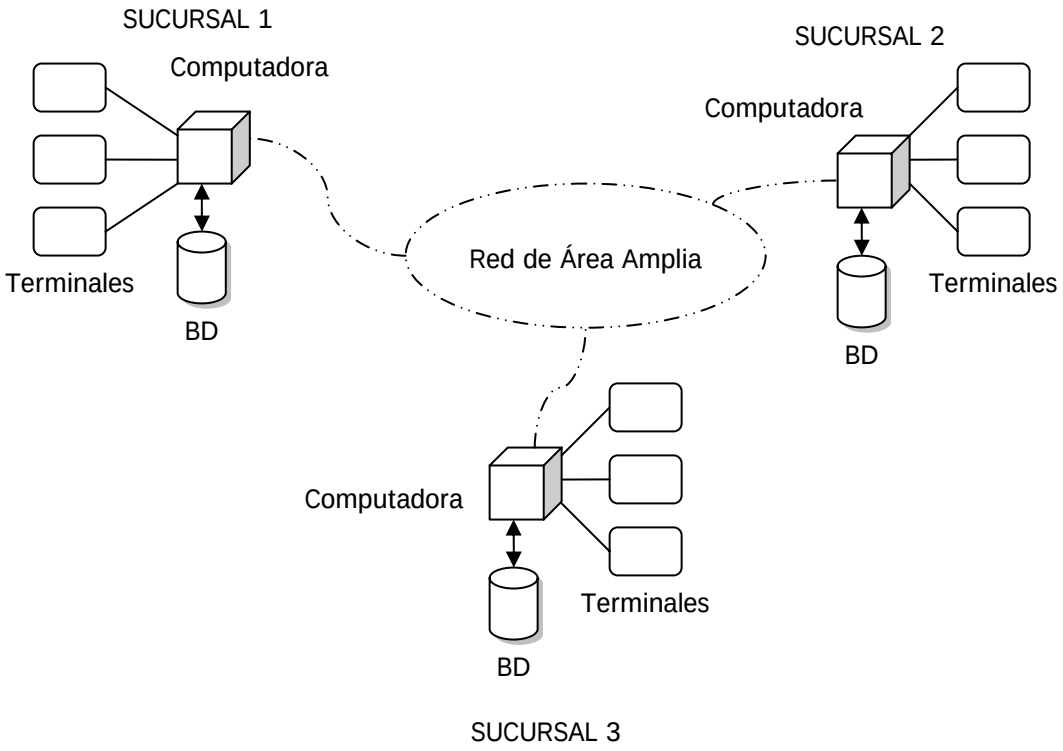



Figura 3. 1. Base de datos distribuida, geográficamente dispersa.

En cada sucursal, una computadora controla las terminales de la misma y el sistema de cuentas. Cada computadora con su sistema de cuentas local en cada sucursal constituye un sitio del SBDD y las computadoras están conectadas por una red de comunicaciones. Durante las operaciones normales, las aplicaciones en las terminales de la sucursal sólo necesitan acceder a la BD local, por lo que se les llaman aplicaciones locales.

Lo más importante de este ejemplo es la existencia de algunas transacciones que acceden a información en más de una sucursal. Estas transacciones son llamadas transacciones globales o transacciones distribuidas. La existencia de transacciones globales será considerada como una característica que nos ayude a discriminar entre los SBDD y un conjunto de base de datos locales.

Una típica transacción global sería una transferencia de fondos de una sucursal a otra. Esta aplicación requiere actualizar datos en dos diferentes sucursales y asegurarse de la real actualización en ambos sitios o en ninguno. Asegurar el buen funcionamiento de aplicaciones globales es una tarea difícil.

En este ejemplo las computadoras están geográficamente en diferentes puntos pero, como se ha mencionado, las BDD también pueden construirse sobre una red local.

Considere ahora el mismo banco, con las mismas aplicaciones, pero con un sistema configurado como se muestra en la figura 3.2.

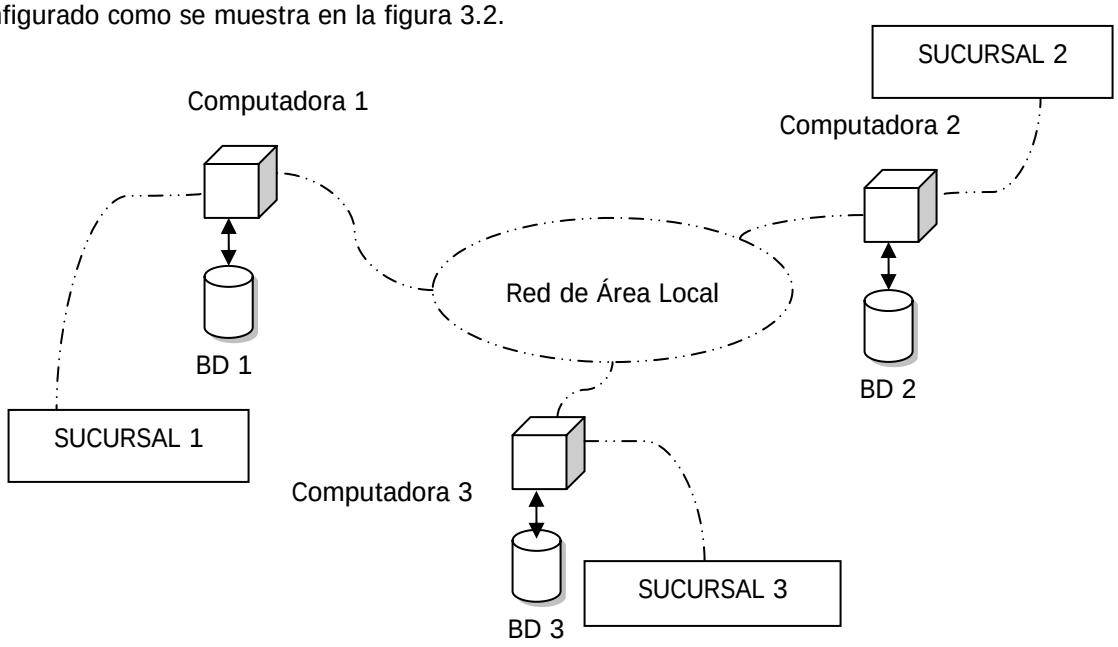



Figura 3.2. Base de datos distribuida en una red de área local.

Los mismos procesadores con sus bases de datos han sido movidos de sus sucursales a un edificio común y ahora están conectados entre sí con un amplio ancho de banda. Las terminales de las sucursales están conectadas a sus respectivas computadoras a través de una red. Cada procesador y su base de datos constituye un sitio de la red local.

Vemos que la estructura física de las conexiones ha cambiado con respecto al ejemplo previo, pero las características de la arquitectura se mantienen. Las mismas computadoras soportan las mismas aplicaciones, accediendo a las mismas bases de datos. La transacción local del ejemplo anterior aún es local, no por el hecho geográfico, si no porque una computadora accede a su propia base de datos en el proceso. En caso de que la aplicación accediera a una base de datos distinta, estaríamos hablando de una transacción distribuida.

Dadas las definiciones y los ejemplos anteriores, es claro que ciertos sistemas no se pueden considerar como SBDD. A continuación mencionamos algunos.

Un sistema de tiempo compartido no incluye necesariamente un sistema de manejo de base de datos y, en caso de que lo haga, éste es generalmente controlado y administrado por una sola computadora.

Un sistema de multiprocesamiento puede administrar una base de datos pero lo hace usualmente a través de un solo sistema manejador de base de datos; los procesadores se utilizan para distribuir la carga de trabajo del sistema completo o incluso del propio SMBD pero actuando sobre una sola base de datos.

Finalmente, una base de datos residente en un solo sitio de una red de computadoras y que es accedida por todos los nodos de la red no es una base de datos distribuida. En realidad, se trata de una base de datos cuyo control y administración está centralizada en un solo nodo pero se permite el acceso a ella a través de la red de computadoras. Generalmente es un sistema cliente-servidor, como se puede apreciar en la figura 3.3.

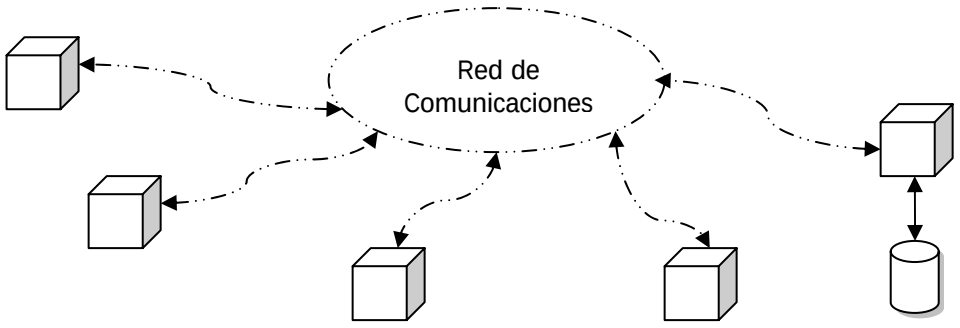



Figura 3.3. Un sistema centralizado sobre una red.

El medio ambiente típico de un SMBDD consiste en un conjunto de sitios que tienen un sistema de procesamiento de datos completo. Así, en diversos nodos se cuenta con una base de datos local, un sistema de manejo de bases de datos y facilidades de comunicaciones. Si los diferentes sitios están geográficamente dispersos, entonces, están interconectados por una red de área amplia (WAN). Por otro lado, si los sitios están localizados en diferentes edificios o departamentos de una misma organización pero geográficamente en la misma ubicación entonces están conectados por una red de área local (LAN). Esto se muestra en la figura 3.4.

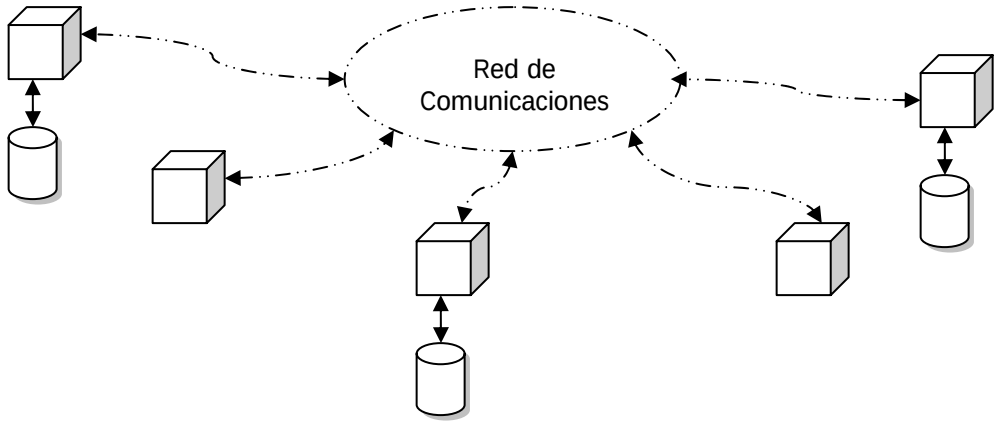



Figura 3.4. Un medio ambiente distribuido para bases de datos.

### 3. 2. Clasificación De Los Sistemas De Base De Datos Distribuida

Desde el punto de vista funcional y de organización de datos, los sistemas de datos distribuidos están divididos en dos clases separadas, basadas en dos filosofías totalmente diferentes y diseñados para satisfacer diferentes necesidades:

- Sistemas homogéneos de base de datos distribuida.
- Sistemas heterogéneos de base de datos distribuida.

Un SMBDD homogéneo integra múltiples recursos de datos, como se muestra en la figura 3.5. Los sistemas homogéneos se parecen a un sistema centralizado, pero en lugar de almacenar todos los datos en un solo lugar, los datos se distribuyen en varios sitios comunicados por una red.

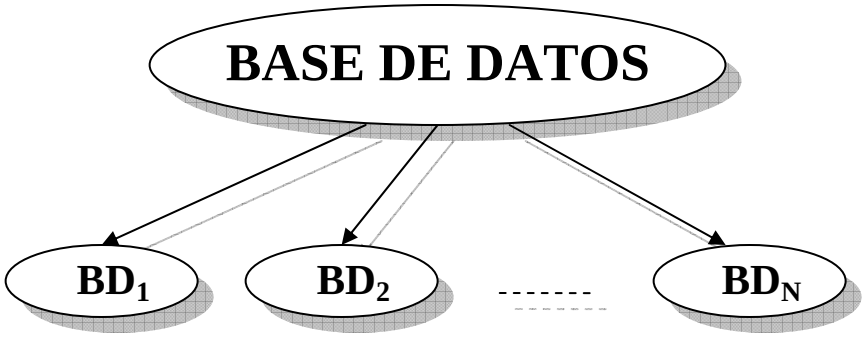



Figura 3.5. Base de datos homogénea:
La base de datos centralizada se distribuye en diversas bases de datos locales.

No existen usuarios locales y todos ellos acceden a la base de datos a través de una interfaz global. El esquema global es la unión de todas las bases de datos locales y el acceso por parte de los usuarios se define sobre el esquema global. Se trata de un conjunto de bases de datos administradas cada una por un mismo SMBD en común.

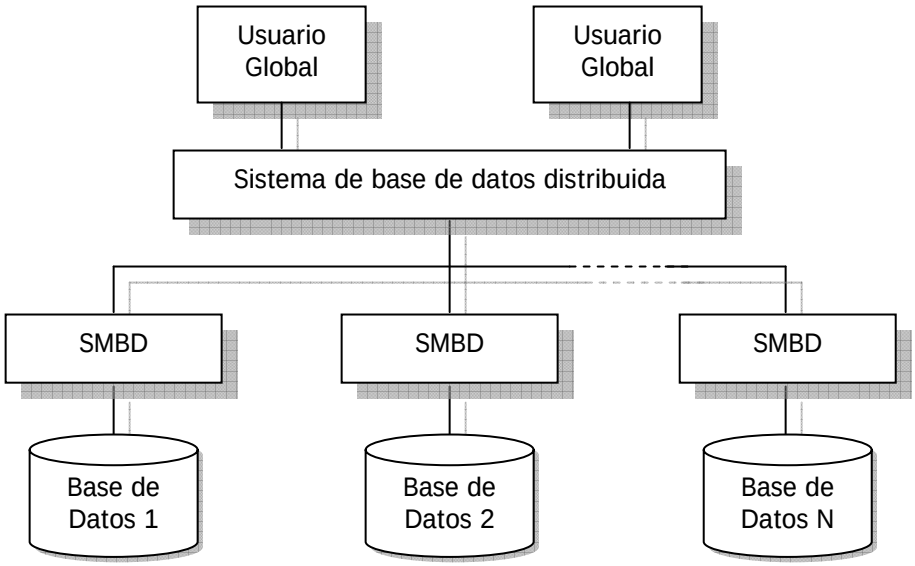



Figura 3.6. Arquitectura de un SMBDD homogéneo.

La clase de sistemas heterogéneos es aquella caracterizada por manejar diferentes SMBD en los nodos locales. Generalmente cada uno de ellos cuenta con un sistema de base de datos local administrado por un SMBD diferente que deberá integrarse a una base de datos distribuida.

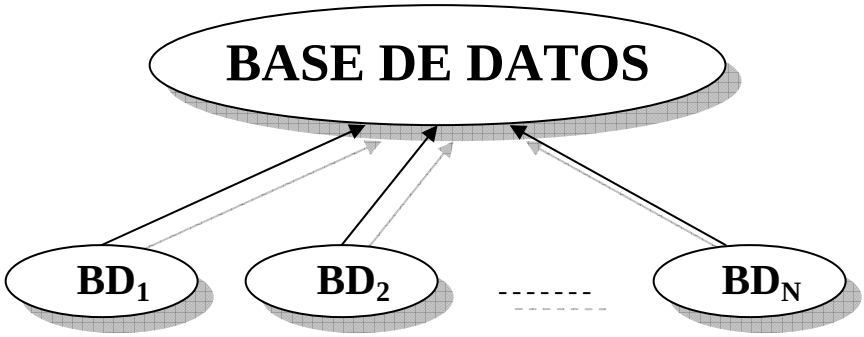



Figura 3.7. Base de datos heterogénea:
Integración de bases de datos existentes en una sola BDD.

Una subclase importante dentro de esta categoría es la de los sistemas de manejo multi-bases de datos. Un sistema multi-bases de datos (Smulti-BD) tiene múltiples sistemas manejadores de bases de datos, que pueden ser de tipos diferentes, y múltiples bases de datos existentes. La integración de todos ellos se realiza mediante subsistemas de software. La arquitectura general de tales sistemas se presenta en la figura 3.8. En contraste con los sistemas homogéneos, existen usuarios locales y globales. Los usuarios locales acceden sus bases de datos locales sin verse afectados por la presencia del Smulti-BD.

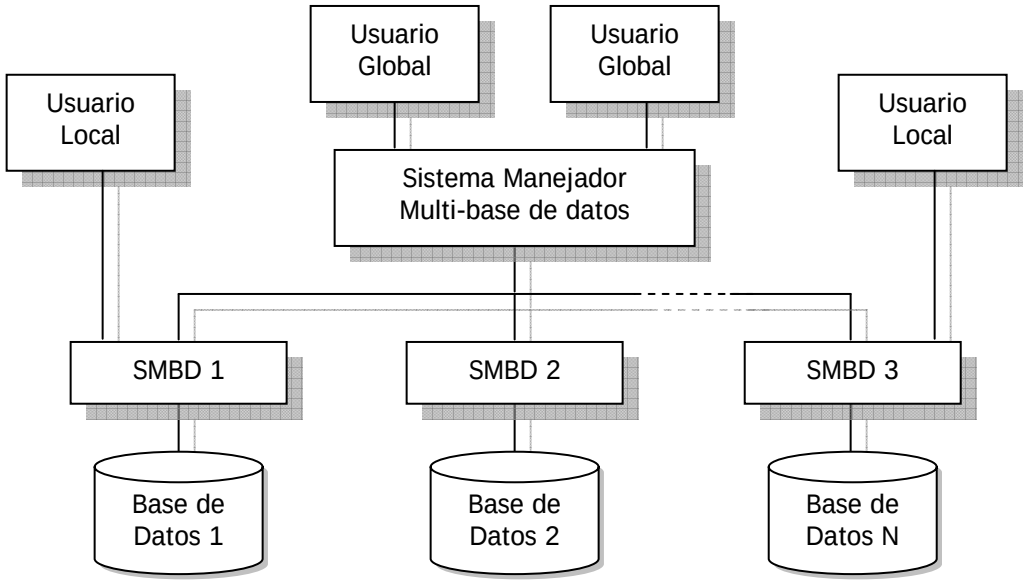



Figura 3.8. Arquitectura de un sistema multi-bases de datos.

## 3. 3. Diseño De Los Sistemas De Base De Datos Distribuida

Con base en la clasificación explicada anteriormente, existen dos estrategias generales para abordar el problema de diseño de bases de datos distribuidas:

### 3. 3. 1. El enfoque de arriba hacia abajo (*top-down*)

Este enfoque es más apropiado para aplicaciones nuevas y, de manera particular, para sistemas homogéneos. Consiste en partir desde un análisis de requerimientos para definir un diseño global. Se prosigue con la fragmentación de la base de datos, y de aquí se continúa con la localización de los fragmentos en los sitios, lo que produce los diferentes esquemas locales de bases de datos. En cada uno de los sitios se cuenta con un sistema manejador de base de datos que accederá a su correspondiente base de datos de manera local. Por conveniencia, todos los manejadores son de un mismo tipo. El conjunto de todas las bases de datos locales constituye la base de datos distribuida.

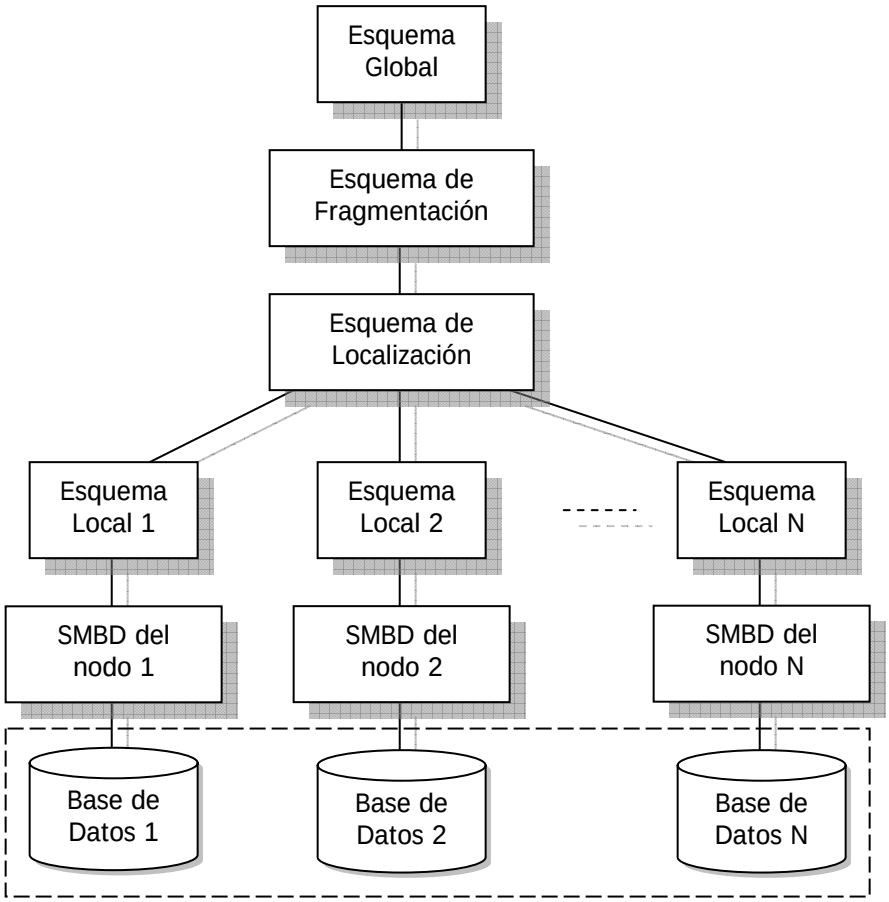



Figura 3.9. Enfoque de arriba hacia abajo

### 3. 3. 2. El enfoque de abajo hacia arriba (*bottom-up*)

Se utiliza particularmente a partir de bases de datos existentes, generando con esto bases de datos distribuidas heterogéneas. En forma resumida, el diseño conceptual global se realiza con base en los esquemas locales ya existentes. Esto se debe a la posibilidad de que se cuente con un SMBD diferente en cada nodo. Las bases de datos que componen al sistema distribuido se encuentran localizadas en diferentes sitios desde un inicio y es necesario integrarlas a un esquema global común.

## 3. 4. Ventajas De Los Sistemas De Bases De Datos Distribuida

- **Mejor rendimiento**: El procesamiento es más rápido debido a que varios nodos intervienen en la realización de una carga de trabajo.

Los datos son localizados cerca del punto de su utilización propiciando que el tiempo de comunicación sea mínimo. Además, cada sitio tiene un mejor número de transacciones en ejecución que si todas las transacciones se enviaran a una sola base de datos centralizada. Varias computadoras operando en forma simultánea pueden entregar más volumen de procesamiento que una sola computadora.

- **Mayor confiabilidad**: La probabilidad de que una falla en un solo nodo afecte al sistema se reduce.

La fiabilidad se define, a grandes rasgos, como la probabilidad de que un sistema esté en funciones en un momento determinado, y la disponibilidad es la probabilidad de que el sistema esté disponible continuamente durante un intervalo de tiempo.

Mediante la replicación de información, las bases de datos distribuidas pueden presentar cierto grado de tolerancia a fallas haciendo que el funcionamiento del sistema no dependa de un solo lugar como en el caso de las bases de datos centralizadas. Cuando falla una computadora, se pueden obtener los datos extraídos de otras computadoras. Los usuarios no dependen de la disponibilidad de una sola fuente de datos.

- **Escalabilidad**: Nuevos nodos se pueden agregar fácil y rápidamente.

Se pueden agregar computadoras adicionales a la red conforme aumenta el número de usuarios y la carga de procesamiento. A menudo es más fácil y más barato agregar una nueva computadora más pequeña que actualizar una computadora única y centralizada. Después, si la carga de trabajo se reduce, el tamaño de la red también puede reducirse.

- **Estructura flexible**: Los sistemas distribuidos se pueden adecuar de una manera más sencilla a las estructuras de organización de los usuarios.

Las razones por las que compañías y negocios migran hacia bases de datos distribuidas incluyen razones organizacionales y económicas, para obtener una interconexión confiable y flexible con las bases de datos existentes, y por un crecimiento futuro. El enfoque distribuido de las bases de datos se adapta más naturalmente a la estructura de las organizaciones. Por ejemplo, una compañía puede tener oficinas en varias ciudades, un banco puede tener múltiples sucursales. Además, la necesidad de desarrollar una aplicación global, que incluya a toda la organización, se resuelve fácilmente con bases de datos distribuidas. Si una organización se expande por medio de la creación de unidades o departamentos nuevos, entonces, el enfoque de bases de datos distribuidas permite un crecimiento suave. Incluso en organizaciones más centralizadas, el procesamiento distribuido ofrece una mayor flexibilidad para adecuarse a la estructura organizacional de lo que permite el procesamiento centralizado.

### 3. 5. Desventajas De Los Sistemas De Base De Datos Distribuida

Las primeras dos ventajas de las bases de datos distribuidas son las mismas que las dos primeras desventajas.

- **Peor rendimiento:** El rendimiento puede ser peor para el procesamiento distribuido que para el procesamiento centralizado.

Depende de la naturaleza de la carga de trabajo, la red, el SMBDD y las estrategias utilizadas de concurrencia y tolerancia a fallas, así como del acceso local a los datos y la interacción de los múltiples procesadores, ya que éstos pueden ser abrumados por las tareas de coordinación y control requeridas. Tal situación es probable cuando la carga de trabajo necesita un gran número de actualizaciones concurrentes sobre datos replicados, y que deben estar muy distribuidos.

- **Menor confiabilidad**: El procesamiento de base de datos distribuida puede resultar menos confiable que el procesamiento centralizado.

De nuevo, depende de la confiabilidad de las computadoras de procesamiento, de la red, del SMBDD, de las transacciones y de las tasas de error en la carga de trabajo. Un sistema distribuido puede estar menos disponible que uno centralizado. Estas dos desventajas indican que un procesamiento distribuido no es ninguna panacea. A pesar de que tiene la promesa de un mejor rendimiento y de una mayor confiabilidad, es necesario realizar un buen análisis y diseño del sistema.

- **Mayor complejidad**: La distribución produce un aumento en la complejidad del diseño y en la implementación del sistema.

Ya que existen más componentes de hardware, hay más cantidad de cosas por aprender y más interfaces susceptibles de fallar. Como los nodos que constituyen el sistema funcionan en paralelo, es más difícil asegurar el funcionamiento correcto de los algoritmos, así como de los procedimientos de recuperación de fallas del sistema. La habilidad para asegurar la integridad de la información en presencia de dichas fallas, en componentes hardware y software, es compleja. El intercambio de mensajes y la ejecución de algoritmos para el mantenimiento de la coordinación entre nodos supone una sobrecarga que no se da en los sistemas centralizados.

- **Mayor costo**: Altos gastos de construcción y mantenimiento.

La complejidad añadida que es necesaria para mantener la coordinación entre nodos hace que el desarrollo de software implique un costo aún mayor. El control de concurrencia y recuperación de fallas puede convertirse en algo complicado y difícil de implementar, puede empujar a una mayor carga sobre programadores y personal de operaciones y quizá se requiera de personal más experimentado.

- **Dificultad de control**: El procesamiento de base de datos distribuido es difícil de controlar.

Esta desventaja se refiere al control y manejo de los datos. Dado que éstos residen en muchos nodos diferentes y se pueden consultar por nodos diversos de la red, la probabilidad de violaciones de seguridad es creciente si no se toman las precauciones debidas. Una computadora centralizada reside en un entorno controlado, y las actividades de procesamiento pueden ser vigiladas, aunque a veces con dificultad. En sistemas distribuidos, en caso de un desastre o catástrofe, la recuperación puede ser más difícil de sincronizar.

| **Ventajas** | **Desventajas** |
|---|---|
| Mejor rendimiento | Peor rendimiento |
| Mayor confiabilidad | Menor confiabilidad |
| Escalabilidad | Mayor complejidad |
| Estructura flexible | Mayor costo |
| | Dificultad de control |

Figura 3.10. Ventajas y desventajas del procesamiento de una base de datos distribuida

### 3. 6. Tipos De Bases De Datos Distribuidas

La figura 3.11 muestra una base de datos no distribuida con cuatro porciones: W, X, Y y Z. Los cuatro segmentos están localizados en una sola base de datos, y no están duplicados.

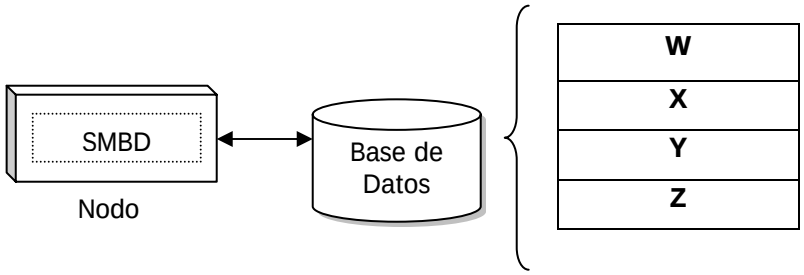



Figura 3.11. Base de datos no distribuida: Alternativa no dividida y no duplicada.

La figura 3.12 muestra la primera alternativa distribuida, en la cual la base de datos ha sido fragmentada en dos porciones: W y X quedan almacenadas en el nodo 1, e Y y Z en el nodo 2.

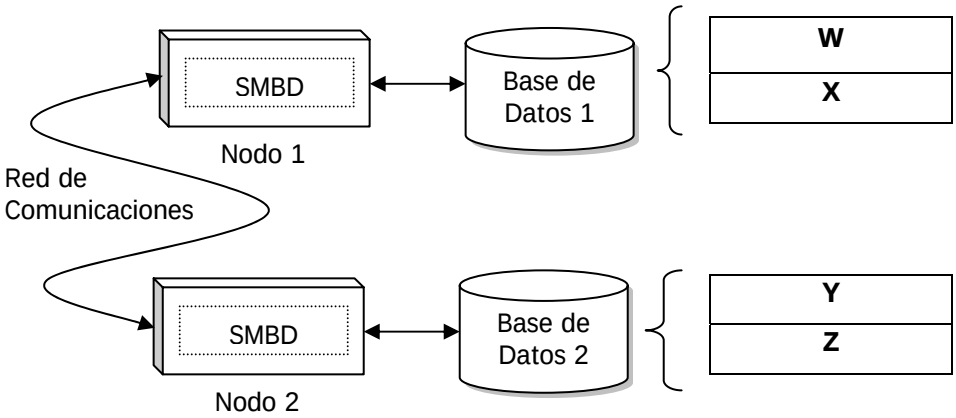



Figura 3.12. Base de datos distribuida: Alternativa fragmentada y no duplicada.

En la figura 3.13 la totalidad de la base de datos ha sido duplicada en los dos nodos.

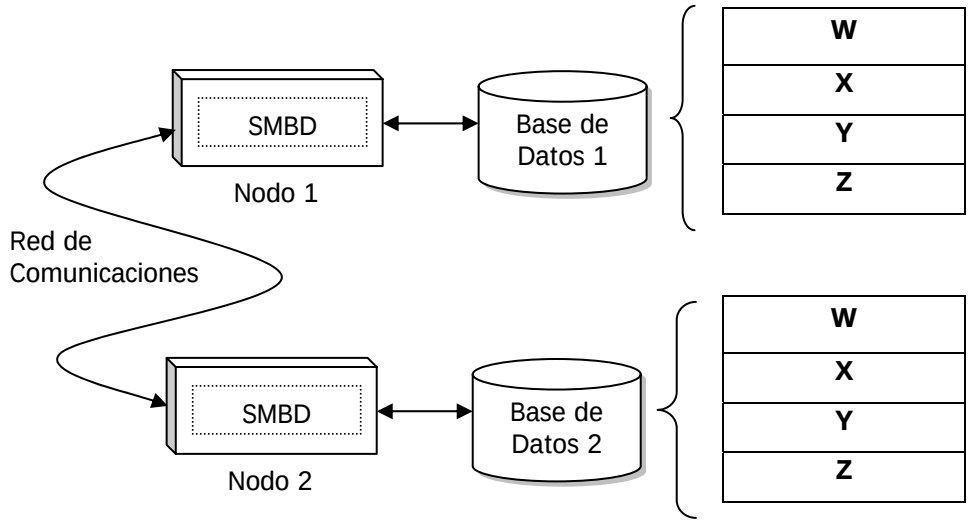



Figura 3.13. Base de datos distribuida: Alternativa duplicada y no fragmentada.

En la figura 3.14 la base de datos ha sido fragmentada, y solamente una porción (Y) ha sido duplicada.

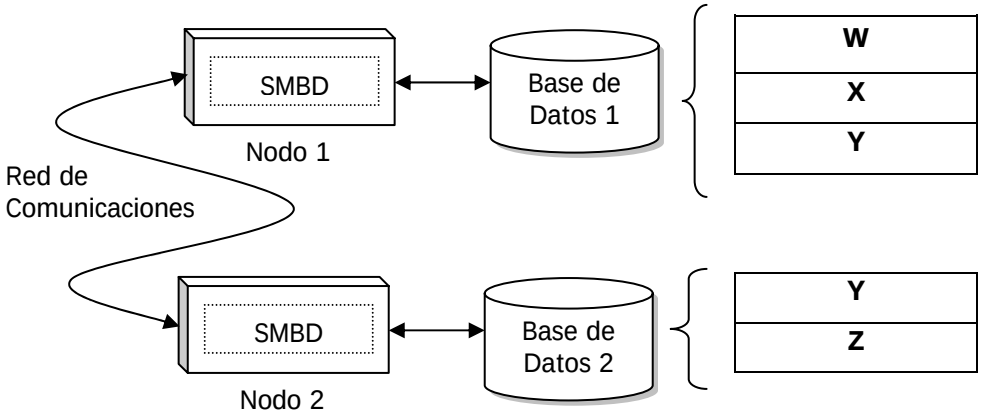



Figura 3.14. Base de datos distribuida: Alternativa fragmentada y duplicada.

### 3. 7. Fragmentación De Una Base De Datos

El problema de fragmentación se refiere al particionamiento de la base de datos para distribuir cada parte en los diferentes sitios de la red. El objetivo de la fragmentación es encontrar un nivel de particionamiento adecuado; desde tuplas o atributos hasta relaciones completas.

Al realizar fragmentaciones de tipo horizontal, vertical o mixta se debe asegurar que no exista pérdida de información así como el hecho de que las relaciones originales puedan obtenerse partiendo de las relaciones fragmentadas.

### 3. 7. 1. Fragmentación Horizontal

Un fragmento horizontal es el resultado de la división de las filas de una relación en diferentes partes. Teniendo una relación R con campos $C_1$, $C_2$, $C_3$, $C_4$, ..., $C_n$ de los cuales $C_1$ es la llave primaria de la relación R($\underline{C_1}$, $C_2$, $C_3$, $C_4$, ..., $C_n$), si las primeras filas o registros se colocan en $R_1$($\underline{C_1}$, $C_2$, $C_3$, $C_4$, ..., $C_n$) y las filas restantes en $R_2$($\underline{C_1}$, $C_2$, $C_3$, $C_4$, ..., $C_n$) se producirán dos fragmentos horizontales.

Considere la relación J del siguiente ejemplo:

**J.** Proyectos que se realizan en una cierta empresa.

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| 1 | Instrumentación | 150000 | Monterrey |
| 2 | Desarrollo de Bases de Datos | 135000 | México |
| 3 | CAD/CAM | 250000 | Puebla |
| 4 | Mantenimiento | 310000 | México |
| 5 | CAD/CAM | 500000 | Guadalajara |

El campo JNO indica las claves de identificación de los proyectos, el campo JNOMBRE indica sus nombres, el campo PRESUPUESTO indica el presupuesto que se ha asignado a su realización, y el campo LUGAR indica las ciudades en las que se desarrollan.

La relación J se puede fragmentar horizontalmente produciendo los siguientes fragmentos:

**J1.** Proyectos con presupuesto menor que $200,000

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| 1 | Instrumentación | 150000 | Monterrey |
| 2 | Desarrollo de bases de datos | 135000 | México |

**J2**. Proyectos con presupuesto mayor o igual a $200,000

| JNO | JNOMBRE | PRESUPUESTO | LUGAR |
|---|---|---|---|
| 3 | CAD/CAM | 250000 | Puebla |
| 4 | Mantenimiento | 310000 | México |
| 5 | CAD/CAM | 500000 | Guadalajara |

Si se sabe que la búsqueda de proyectos con presupuesto mayor o igual a $200,000 se da de manera frecuente, esta fragmentación agilizaría la búsqueda.

### 3. 7. 2. Fragmentación Vertical

Un fragmento vertical es el resultado de la división de una tabla en dos o más conjuntos de columnas. Por lo tanto una relación $R(\underline{C_1}, C_2, C_3, C_4, ..., C_n)$ puede separarse en dos fragmentos verticales $R_1(\underline{C_1}, C_2)$ y $R_2(C_3, C_4, ..., C_n)$. Dependiendo de la aplicación y de las razones para la creación de las divisiones, lo más probable es que también se coloque la llave primaria de R en $R_2$ para formar $R_2(\underline{C_1}, C_3, C_4, ..., C_n)$.

La relación J también puede fragmentarse verticalmente, produciendo los siguientes fragmentos:

**J1**. Información acerca de presupuestos de proyectos

| JNO | PRESUPUESTO |
|---|---|
| 1 | 150000 |
| 2 | 135000 |
| 3 | 250000 |
| 4 | 310000 |
| 5 | 500000 |

**J2**. Información acerca de nombres y ubicaciones de los proyectos

| JNO | JNOMBRE | LUGAR |
|---|---|---|
| 1 | Instrumentación | Monterrey |
| 2 | Desarrollo de bases de datos | México |
| 3 | CAD/CAM | Puebla |
| 4 | Mantenimiento | México |
| 5 | CAD/CAM | Guadalajara |

Debe notarse que en este caso cada una de las relaciones generadas por la fragmentación contiene la llave primaria de la relación original para identificar la información relacionada entre fragmentos.

Algunas veces se separa una relación en divisiones horizontales y después verticales o viceversa, y al resultado se le llama fragmentación mixta.

### 3. 8. Comparación De Las Alternativas De Bases De Datos Distribuidas

Los diferentes tipos de bases de datos distribuidas se resumen en un continuo en la figura 3.15, organizado en un grado creciente de distribución, de izquierda a derecha. La base de datos no distribuida aparece en el punto más a la izquierda del continuo, y la base dividida y fragmentada en el punto más a la derecha. Entre estos extremos aparece la base de datos con particiones, en la cual las particiones se asignan a dos o más computadoras. También se muestra una base de datos que no está dividida pero sí replicada, con lo que cada base de datos completa queda duplicada en dos o más computadoras.

La figura 3.15 también muestra las características de las alternativas mencionadas. Las que se encuentran hacia la derecha incrementan el paralelismo, la independencia, la flexibilidad y la disponibilidad; significan también un gasto mayor, una complejidad mayor, un grado mayor de dificultad de control, y un riesgo mayor en relación con la seguridad.

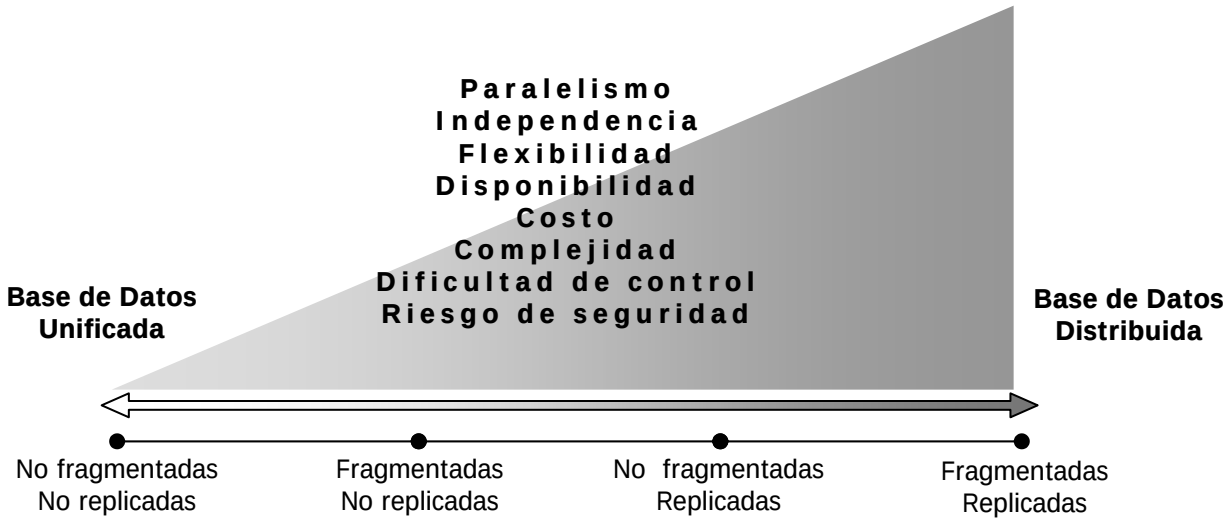



Figura 3.15. Contínuo de alternativas de base de datos.

Las alternativas a la derecha proporcionan mayor flexibilidad, y pueden adecuarse mejor a la estructura y al proceso organizacional de una institución. Una empresa manufacturera descentralizada, en la cual los gerentes de planta tienen una amplia libertad en su planeación, difícilmente quedarán satisfechos con un sistema de información que tenga una estructura

centralizada porque la arquitectura del sistema de información y la estructura de la empresa chocan entre sí. Las alternativas del lado derecho proporcionan un esquema mejor y más apropiado a esta organización que las del lado izquierdo.

### 3. 9. Procesamiento De Base De Datos Distribuida

El procesamiento de base de datos distribuida es aquel procesamiento de base de datos en el cual la ejecución de transacciones y la recuperación y actualización de datos acontece a través de dos o más computadoras independientes, por lo general separadas geográficamente pero enlazadas a través de una red.

La figura 3.16 muestra un sistema de base de datos distribuida que involucra cuatro computadoras.

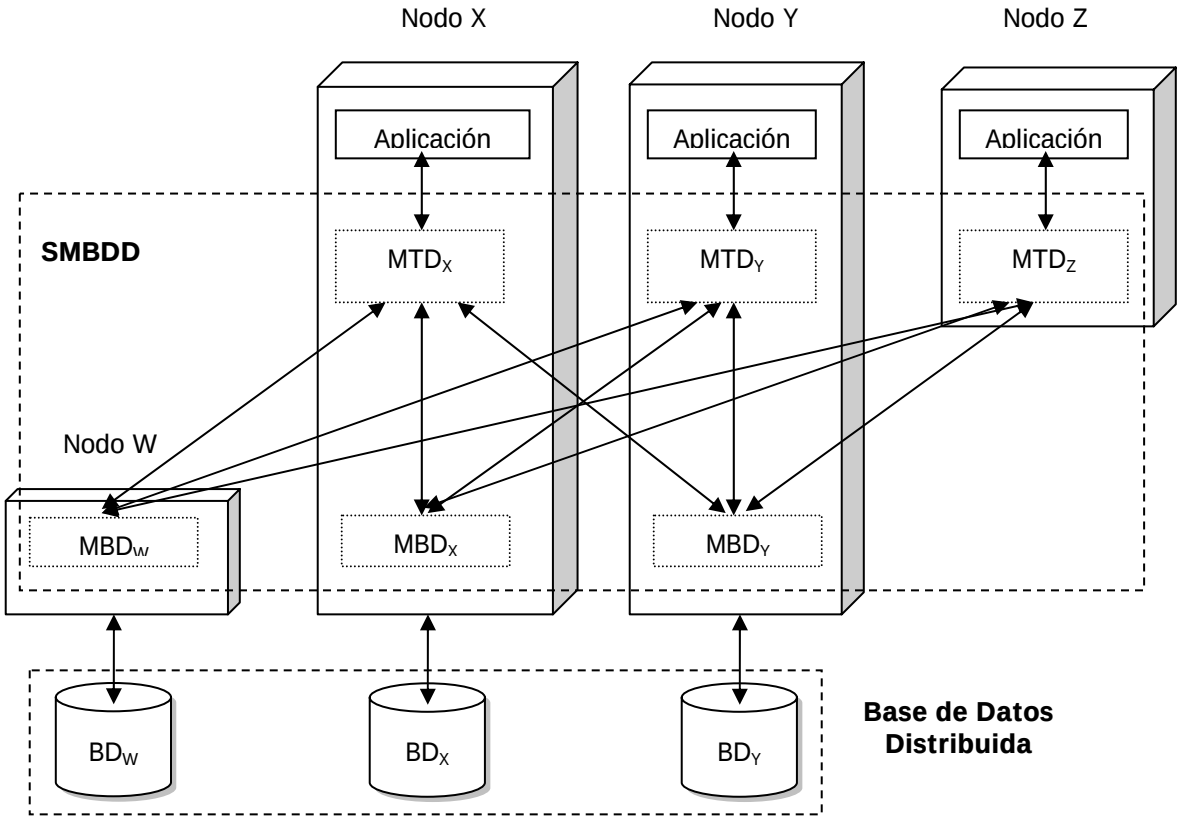



Figura 3.16. Procesamiento de base de datos distribuida.

El sistema manejador de base de datos distribuida (SMBDD), está formado por las transacciones y los manejadores de base de datos de todas las computadoras. Tal y como se muestra, el SMBDD es un esquema genérico que implica un conjunto de programas que operan en

diversos sitios. Estos programas pueden ser el subsistema de un producto único SMBDD, concesionado por un solo fabricante, o también pudiera resultar una colección de programas de fuentes dispares: algunos concesionados por fabricantes y algunos otros desarrollados de manera particular. El propósito de la figura 3.16 es ilustrar las funciones que deben atenderse en el procesamiento de bases de datos distribuidas.

Un manejador de transacciones distribuidas (MTD) es un programa que recibe solicitudes de procesamiento de los programas de consulta y las traduce en acciones para los manejadores de base de datos locales. Una función importante del MTD es coordinar y controlar dichas acciones. Dependiendo de la naturaleza de la aplicación, el MTD puede ser proporcionado como parte de un SMBDD comercial o puede desarrollarse por quien pone en práctica el sistema distribuido. En aplicaciones menos complejas, una parte de sus funciones puede ser llevada a cabo por personas, siguiendo procedimientos manuales.

Un manejador de base de datos (MBD) es un programa que procesa cierta porción de la base de datos distribuida, como es el hecho de recuperar y actualizar datos en alguno de los nodos, de acuerdo con comandos de acción recibidos de los MTD. El MBD puede ser un subconjunto de un producto SMBDD, o ser también un SMBD comercial no distribuido. En algunos casos, el SMBDD puede contener diferentes productos SMBD locales.

Un nodo es una computadora que ejecuta un MTD, un MBD, o inclusive ambos. Un nodo de transacción procesa un MTD, y un nodo de base de datos procesa un MBD y su base de datos. En la figura 3.16 el nodo W es un nodo de base de datos ejecutando $MBD_W$ y almacenando $BD_W$. El nodo X es tanto un nodo de transacción como de base de datos con MTDx, MBDx y BDx. De modo similar, el nodo Y es tanto un nodo de transacción como de base de datos, pero el nodo Z es solamente un nodo de transacción.

Las aplicaciones o los programas de consulta solicitan datos a los MTD y éstos, a su vez, realizan solicitudes de acción a los MBD. Un ejemplo es INSERT empleado= ”Juan Pérez”. Este tipo de solicitudes operan sobre estructuras lógicas. El programa de consulta o de aplicación no se refiere a ninguna instancia física en particular de la estructura de la base de datos.

Los MTD se comunican con los MBD por medio de acciones a ejecutarse en ocurrencias específicas de datos. Por lo tanto, si la nueva ocurrencia de INSERT empleado= ”Juan Pérez” debe realizarse en BDx y en BDy, el MTD traducirá la solicitud en dos acciones. Una se dirigirá a MBDx

para almacenar los nuevos datos, y la segunda se dirigirá a MBDy para, a su vez, almacenar tal información.

### 3. 10. Procesamiento De Búsquedas

El procesamiento de búsquedas es el medio por el cual se manejan las búsquedas en un sistema distribuido. Cuando las búsquedas operan sobre datos locales son manejadas de la misma forma que en las bases de datos centralizadas. Sin embargo, existen dos maneras en las que pueden manejarse las búsquedas cuando intervienen datos de sitios externos.

El primer método se enfoca en el nodo que obtiene datos de otros sitios del sistema. Los conjuntos de datos son enviados al nodo que los requiere y son procesados de manera local, de manera similar a una arquitectura de recursos compartidos. Este método es simple pero consume tiempo en dos áreas. Primero, todos los datos deben ser transferidos a través de la red, lo cual puede implicar un gran consumo de recursos. Segundo, el nodo que inicia la búsqueda es el responsable de realizar todas las operaciones, lo que puede requerir una gran cantidad de tiempo de procesador.

El segundo método propone que el nodo que realiza la consulta distribuya las operaciones de búsqueda a los demás sitios para que éstas sean efectuadas remotamente en cada sitio donde se encuentren los datos de interés y así obtener un resultado, de manera similar a lo que ocurre en un sistema cliente-servidor. El tráfico en la red es reducido ya que no se devuelven datos innecesarios al nodo inicial. El poder de procesamiento en máquinas remotas es utilizado con base en la carga de trabajo que requiere dicho nodo. Éste debe poseer la capacidad de optimizar la búsqueda y de distribuirla a los sitios remotos para obtener el conjunto de datos deseado.

Los sistemas de base de datos distribuida deben contar con control de recuperación. Cuando se realizan actualizaciones a una base de datos se debe hacer con una filosofía todo o nada. En caso de que se produzca un error en la actualización de algún dato, el sistema debe restaurar el estado de la base de datos a aquél en el que se encontraba antes de que la operación comenzara.

El control de concurrencia es una parte importante del control de recuperación. Este tipo de control es la habilidad para probar y establecer candados en todos los niveles lógicos dentro de una

base de datos. Esta área difiere de un sistema de base de datos tradicional en que los mensajes deben enviarse a través de la red para coordinar una transacción entre nodos.

## 3. 11. Componentes De Un Sistema De Base De Datos Distribuida

Existen muy diferentes tipos de procesamiento que son englobados por el término procesamiento de base de datos distribuida, siendo todos válidos de acuerdo con la arquitectura general mostrada en la figura 3.16. Así, cada base de datos distribuida tiene sus propias capacidades y su propia serie de problemas. Sin embargo, los siguientes elementos pueden considerarse como parte fundamental de cualquier base de datos distribuida.

- **Hardware**

En algunos sistemas distribuidos todos los nodos son homogéneos, es decir, cuentan con las mismas características tanto en hardware como en sistema operativo, programas de aplicación y sistema local de base de datos, sólo por mencionar algunos aspectos. Otros sistemas cuentan con nodos heterogéneos. Las diferencias en las velocidades de procesamiento y las capacidades de almacenamiento deben considerarse al determinar los roles que jugarán los nodos en el SBDD.

- **Programas**

El programa principal que se debe considerar en un SBDD es el SMBDD. La arquitectura de un SMBDD mostrada en la figura 3.16 es genérica. Los MTD y los MBD pueden ser subsistemas de un único SMBDD adquirido como producto. De manera alternativa y más frecuente, el SMBDD puede ser un conjunto de programas desarrollados de manera particular y de productos desarrollados por compañías de software. En muchos casos, incluido el presente proyecto de tesis, los MTD son escritos de manera particular y los MBD son SMBD comerciales.

- **Datos**

Una de las partes clave de un SBDD es la replicación de datos. Una BDD puede no estar replicada, estar parcialmente replicada o totalmente replicada. Si no está replicada, sólo existe una copia de cada elemento de la base de datos. Estos elementos son asignados a nodos particulares y sólo pueden residir en su nodo correspondiente. Cualquier aplicación que necesite acceder a un elemento debe obtenerlo del nodo oficialmente designado.

Una base de datos parcialmente replicada contiene algunos elementos duplicados y otros no. Para tal efecto debe existir un directorio que indique si un determinado elemento de la base está replicado y dónde está almacenado. Una base de datos totalmente replicada es aquella en la que la base de datos entera es duplicada en dos o más nodos.

- **Procedimientos**

Existe una multitud de componentes procedurales de SBDD. El primer grupo de procedimientos concierne a los derechos de procesamiento. En general, cualquier nodo puede llevar a cabo una petición de actualización para cualquier elemento de la base de datos en su nodo o en un nodo distinto. Si el elemento está replicado, todas las copias deben actualizarse y entre más aplicaciones actualicen datos replicados, más costoso será el sistema.

Un componente procedural más concierne al control. En caso de que ocurra un conflicto entre distintas peticiones de procesamiento de las base de datos , ¿qué nodo resuelve el conflicto? En general, un control centralizado es más fácil de implementar que aquel basado en la igualdad de nodos, donde ningún nodo está a cargo y las decisiones de control son tomadas por cualquier nodo, dependiendo del problema y del estado del sistema. Este esquema permite una mayor flexibilidad pero es mucho más complicado.

- **Personal**

Los sistemas distribuidos varían considerablemente en las demandas que sitúan sobre las personas. Aquellos sistemas con un SMBDD sofisticado y poderoso requieren poca intervención de los usuarios. De hecho, los usuarios no saben que están procesando datos distribuidos. Simplemente acceden sus aplicaciones y todos los procesos de distribución son manejados por el SMBDD. Para aquellos sistemas menos sofisticados, los usuarios deben involucrarse.

Dependiendo del diseño del sistema, los usuarios pueden tener la responsabilidad de inspeccionar reportes del procesamiento que el sistema realiza para determinar que los datos sean transferidos correctamente.

En SBDD muy elementales, los usuarios deben cargar con algunas responsabilidades que típicamente corresponden al MTD. Por ejemplo, en algunos sistemas los usuarios deben hacer cambios a la base de datos local y, de manera manual, hacer efectivos dichos cambios para los datos replicados en nodos ajenos.

### 3. 12. Tipos de fallas en un SMBDD.

Diseñar un sistema confiable que se pueda recuperar de fallas requiere identificar los tipos de fallas con las cuales el sistema tiene que tratar. Así, los tipos de fallas que pueden ocurrir en un SMBDD son:

- **Fallas de transacciones**. Las fallas en transacciones se pueden deber a un error en los datos de entrada o a la ocurrencia de un interbloqueo. La forma usual de enfrentar las fallas en transacciones es abortarlas.

- **Fallas del sistema**. En un sistema distribuido se pueden presentar fallas en el procesador, la memoria principal o la fuente de energía de un nodo. En este tipo de fallas se asume que el contenido de la memoria principal se pierde, pero el contenido del almacenamiento secundario es seguro. Típicamente, se hace diferencia entre las fallas parciales y fallas totales. Una falla total se presenta en todos los nodos del sistema distribuido. Una falla parcial sólo se presenta en algunos nodos del sistema.

- **Fallas del medio de almacenamiento**. Se refiere a las fallas que se pueden presentar en los dispositivos de almacenamiento secundario que contienen las bases de datos. Estas fallas se pueden presentar por errores del sistema operativo, por errores en la base de datos, o en el disco duro.

- **Fallas de comunicación**. Las fallas de comunicación en un sistema distribuido son frecuentes. Éstas se pueden manifestar como pérdida de mensajes; lo que lleva, en un caso extremo, a dividir la red en varias subredes separadas.

### 3. 13. Metas Para Un SMBDD

El conocer las metas que debe cumplir un sistema de base de datos distribuida brinda un excelente marco de referencia para una investigación de los temas, problemas y soluciones propuestas para dichos sistemas. Las metas más importantes involucran aspectos de transparencia.

En un sistema de base de datos distribuida la transparencia significa que las opciones de consulta y los programas de transacciones quedan aislados de la administración de la base de datos distribuida, de forma tal que obtengan las ventajas del procesamiento distribuido, sin por ello involucrarse en los detalles de la distribución de la base de datos. Los programadores y los usuarios

pueden concentrarse en la naturaleza y en la lógica de su aplicación, sin verse obligados a tratar asuntos que corresponden al SMBDD.

Las transacciones necesitan tener acceso a la base de datos a través de un SMBDD que proporcione los cuatro siguientes tipos de transparencia: Localización de datos, duplicación de datos, concurrencia y falla.

- **Transparencia de localización**

Las transacciones necesitan ser independientes de la localización particular de los datos distribuidos. De no ser así, las cuestiones de localización complicarían la lógica de una transacción. Considere usted una empresa manufacturera donde el gerente de inventarios desea mover refrigeradores de la planta A a la planta B y que, para lograrlo, deberán modificarse dos registros de inventario. Suponga que los datos involucrados no están duplicados, pero que puedan estar almacenados en una computadora en cualquiera de dos localizaciones. Si el programa que procesa esta transacción no es transparente en lo que se refiere a localización de los datos, tendrá que considerar cuatro casos: ambos registros en A, uno en A y uno en B, uno en B y el otro en A o ambos en B. La lógica de la transacción se complica por la necesidad de considerar la localización de los datos.

Se puede conseguir la transparencia de localización si los MTD son responsables de determinar la localización de los datos y de emitir las acciones a los MBD apropiados, lo cual puede realizarse si los MTD poseen acceso a los directorios de localización de los datos. Si éstos se mueven, sólo el MTD necesita involucrarse. Todas las transacciones quedan aisladas de la modificación.

- **Transparencia de duplicación**

Las transacciones son optimizadas cuando no son responsables de los datos duplicados. Una transacción puede actuar como si éstos estuvieran almacenados una sola vez en un único nodo. Con la transparencia de duplicación, se pueden crear nuevos duplicados, o los duplicados existentes pueden ser eliminados, sin afectar las transacciones de los usuarios o el procesamiento de las consultas.

En una lectura de datos, el MTD selecciona uno de los nodos que los almacena y emite una acción de consulta. Para facilitar la selección, el MTD podría conservar estadísticas sobre el tiempo que se requiere para leer datos duplicados en varios nodos, y seleccionar el nodo con el mejor rendimiento. Es más complicada la escritura de datos duplicados, porque el MTD deberá emitir una acción de escritura para cada uno de los MBD que almacena una copia de dichos datos.

Este análisis supone que cada MTD posee una copia exacta y actualizada de un directorio de localización. No obstante, es necesario un alto grado de coordinación si consideramos lo que ocurriría cuando el directorio sea modificado para tomar en cuenta nuevas copias de datos o su eliminación. Todos los directorios deben actualizarse de forma que ningún MTD determine que los datos están disponibles antes de que así sea. De lo contrario, un MTD podría solicitar datos que todavía no existan o dejar de emitir una orden de escritura a un MBD.

- **Transparencia de concurrencia**

Aunque múltiples transacciones que involucran la base de datos distribuida se lleven a cabo al mismo tiempo, el resultado total de las transacciones no deberá cambiar. El SMBDD proporciona transparencia de concurrencia si los resultados de todas las transacciones concurrentes son consistentes de manera lógica con los resultados que se habrían obtenido si las transacciones se hubieran ejecutado una por una, en algún orden serial arbitrario. La lógica de las transacciones procesadas en forma concurrente deberá ser la misma que si las transacciones se hubieran procesado de manera secuencial.

- **Transparencia de fallas**

La cuarta meta de un SMBDD es proporcionar transparencia de fallas, lo que implica el correcto funcionamiento del sistema a pesar de las fallas ocurridas en una transacción, en el SMBDD, en la red o en algún nodo. Frente a una falla, las transacciones deberán ser atómicas, esto es, ya sea que se procesen todas sus instrucciones o ninguna de ellas. Además, las modificaciones efectuadas por una transacción deben ser permanentes.

Para lograr las metas antes citadas, el software de SMBDD debe contar con las funciones de un SMBD centralizado y además con las siguientes:

- La capacidad de tener acceso a sitios remotos y transmitir consultas y datos entre los diversos sitios a través de una red de comunicaciones.

- La capacidad de seguir la pista de la distribución y la replicación de los datos.

- La capacidad de elaborar estrategias de ejecución para consultas y transacciones que tiene acceso a datos a más de un sitio.

- La capacidad de recuperarse de caídas de sitios individuales y de fallas en un enlace de comunicación.

# CAPÍTULO 4
# El Cómputo Distribuido Con Java RMI

Los sistemas distribuidos requieren que las partes que los componen y que se ejecutan en diferentes espacios de direcciones (posiblemente en computadoras independientes), tengan la capacidad de comunicarse y compartir datos y secuencias de instrucciones.

Una de las primeras soluciones a esta problemática fueron los *sockets*, que precisamente tienen la capacidad de comunicar dos procesos. Sin embargo, los *sockets* requieren que las aplicaciones implementen sus propios protocolos para codificar y decodificar los mensajes que intercambian, lo que introduce un grado de complejidad mayor en el problema a resolver y aumenta la posibilidad de errores durante la ejecución.

Debido a la necesidad de abstraer la comunicación entre procesos, surgieron las llamadas a procedimientos remotos (*Remote Procedure Calls* o RPC), donde la comunicación entre los elementos que componen el sistema distribuido se realiza mediante la invocación de funciones que se encuentran en espacios de direcciones diferentes. En este caso, el programador tiene la impresión de trabajar con procedimientos locales, mientras que en realidad el sistema RPC se encarga de empaquetar los argumentos y enviarlos al proceso que contiene el código que implementa a la rutina remota, en alguna otra computadora.

Con base en las llamadas a procedimientos remotos, se construyeron los sistemas distribuidos orientados a objetos, que proveen los mecanismos necesarios para un manejo transparente de esta comunicación, de modo que el programador sólo se ocupe de la lógica de su aplicación.

## 4. 1. Los Sistemas Distribuidos Orientados a Objetos

Dentro de la programación orientada a objetos, un objeto puede definirse como una estructura de datos y conjunto de procedimientos que los manipulan. El acceso y la modificación de los datos de un objeto se realiza solamente a través de estos procedimientos, llamados métodos. Los objetos se crean y eliminan durante la ejecución de un programa y pueden interactuar con otros objetos.

Un sistema orientado a objetos consta de una colección de objetos que interactúan entre sí, solicitando y proporcionando datos mediante la invocación de sus métodos, por lo que se ajusta bastante bien al diseño de sistemas distribuidos, debido a que las invocaciones de métodos pueden extrapolarse a invocaciones sobre métodos de objetos localizados en máquinas diferentes a través de una red.

En el caso de un sistema de objetos distribuidos, los principales requisitos son la habilidad de crear e invocar objetos en una máquina o proceso remoto, e interactuar con ellos como si se tratara de objetos dentro de un mismo proceso local. Es necesario contar con un protocolo de intercambio de mensajes para enviar peticiones de creación de objetos remotos, invocación de métodos y eliminación de dichos objetos. Los objetos distribuidos radican en aplicaciones que pueden actuar como clientes, servidores o ambos, dependiendo de si contienen objetos remotos o referencias a ellos.

Para crear un objeto remoto en un sistema de objetos distribuidos, es necesario hacer referencia a una clase remota, brindar argumentos para su método constructor y recibir una referencia al objeto que la clase ha creado. Dicha referencia será utilizada para invocar métodos sobre el objeto y, en algún momento, eliminarlo cuando hayamos dejado de usarlo. Así, los datos que debemos enviar a través de la red incluyen referencias a clases, referencias a objetos, referencias a métodos y argumentos de métodos.

Los sistemas de objetos distribuidos son la base de las arquitecturas modernas de tres capas. En una arquitectura de tres capas, como la mostrada en la figura 4.1., la lógica de presentación reside en el cliente (primer capa), la lógica de negocio en el servidor (la capa de en medio o segunda capa), y otros recursos, como una base de datos, residen en la tercer capa.

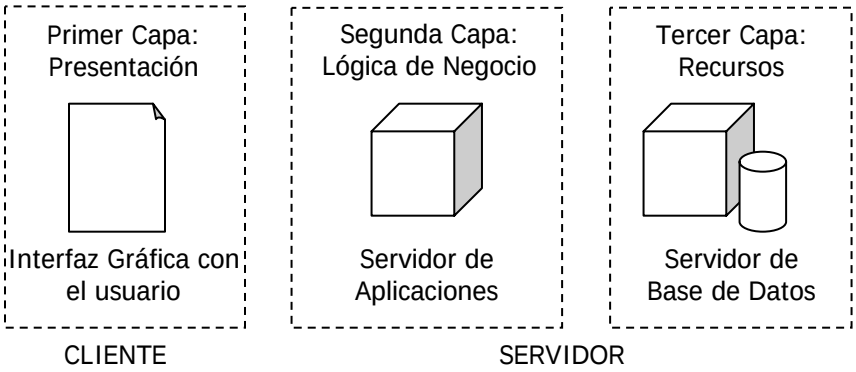



Figura 4.1. Arquitectura de tres capas.

Todos los protocolos de objetos distribuidos se basan en la misma arquitectura, que es diseñada para hacer que un objeto ubicado en una computadora parezca residir en otra. Las arquitecturas de objetos distribuidos se basan en una capa de comunicaciones de red que es muy simple. Esencialmente, existen tres partes: un programa llamado servidor de objetos, un objeto llamado *skeleton* y un objeto llamado *stub*.

El *stub* y el *skeleton* son responsables de hacer que el servidor de objetos, que reside en el servidor, parezca ejecutarse localmente en la máquina cliente. Esto se logra a través de algún tipo de protocolo de Invocación Remota de Métodos (RMI).

### 4.1.1. El protocolo de Invocación Remota de Métodos

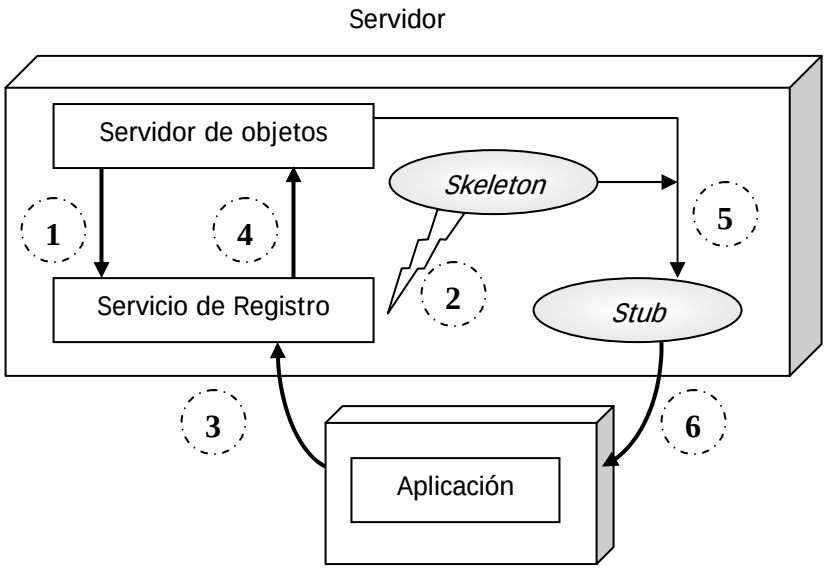
El protocolo RMI (*Remote Method Invocation*) es utilizado por diferentes aplicaciones para comunicar invocaciones de métodos a través de una red, permitiendo la creación de sistemas distribuidos orientados a objetos. En esta sección describiremos el funcionamiento de dicho protocolo independientemente del lenguaje de programación que lo implemente. Por citar algunos ejemplos, tecnologías como CORBA, Java y Microsoft DCOM realizan sus propias implementaciones de dicho protocolo.

Las aplicaciones basadas en el protocolo RMI generalmente están compuestas de dos programas separados: un cliente y un servidor. Un programa servidor típico crea algunos objetos remotos y referencias a ellos, hace que dichas referencias estén disponibles para ser usadas por sus clientes y espera a que alguno de ellos invoque métodos sobre ellas. Por otra parte, un programa cliente típico obtiene una referencia a un objeto remoto ubicado en el servidor e invoca sus métodos.

Para evitar confusiones al referirnos al programa servidor o al nodo servidor donde se están ejecutando los objetos remotos, llamaremos *servidor de objetos* al programa que se está ejecutando en el nodo servidor, y a éste lo seguiremos llamando simplemente *servidor*.

Para que sus métodos estén disponibles a los clientes, el servidor de objetos puede utilizar un servicio de registro. Éste es un proceso que se ejecuta en segundo plano y mantiene la información de aquellas clases que brindan objetos remotos. También es responsable de asociar un nombre único a cada una de dichas clases y generar una referencia (*skeleton*) lista para utilizarse.

Ya que el servidor de objetos cuenta con una clase totalmente registrada, el cliente puede solicitar un objeto remoto de dicha clase a través del servicio de registro, buscando el objeto que necesita por el nombre con el que fue registrado. El servicio de registro dirige la petición del cliente a la clase correspondiente, que crea e inicializa el nuevo objeto remoto utilizando el *skeleton* previamente almacenado. Tal objeto es ahora cargado en memoria por el servidor, y una referencia a él es devuelta al cliente en forma de un objeto *stub*. Éste es utilizado por el cliente para interactuar con el objeto remoto que se encuentra en el servidor. Las transacciones involucradas en la petición y el uso de un objeto remoto se muestran en la figura 4.2.




**1**. El servidor de objetos se registra en el Servicio de Registro.

**2.** El Servicio de Registro asigna un nombre al servidor de objetos y crea un *skeleton*.

**3**. La aplicación en el cliente solicita un objeto remoto al servicio de registro.

**4**. El servicio de registro canaliza la petición a la clase registrada con el nombre solicitado.

**5**. La clase registrada crea un objeto, lo carga en memoria, le asigna el *skeleton* previamente almacenado y crea otra referencia llamada *stub*.

**6**. El objeto *stub* es enviado al cliente como la representación del objeto remoto solicitado.

Figura 4.2. Solicitud y obtención de un objeto remoto en el protocolo RMI.

Cada instancia del servidor de objetos es encapsulada por una instancia de su

correspondiente *skeleton*. A partir de ese momento, el *skeleton* se encuentra escuchando por algún puerto del servidor, en espera de que el *stub* obtenido por el cliente invoque algún método remoto.

El *stub* reside en la máquina cliente y está conectado con el *skeleton* a través de la red. El *stub* actúa como una representación del servidor de objetos en el cliente y es responsable de comunicar las peticiones que una aplicación haga al servidor de objetos por medio del *skeleton*. La figura 4.3. ilustra la invocación de un método remoto.

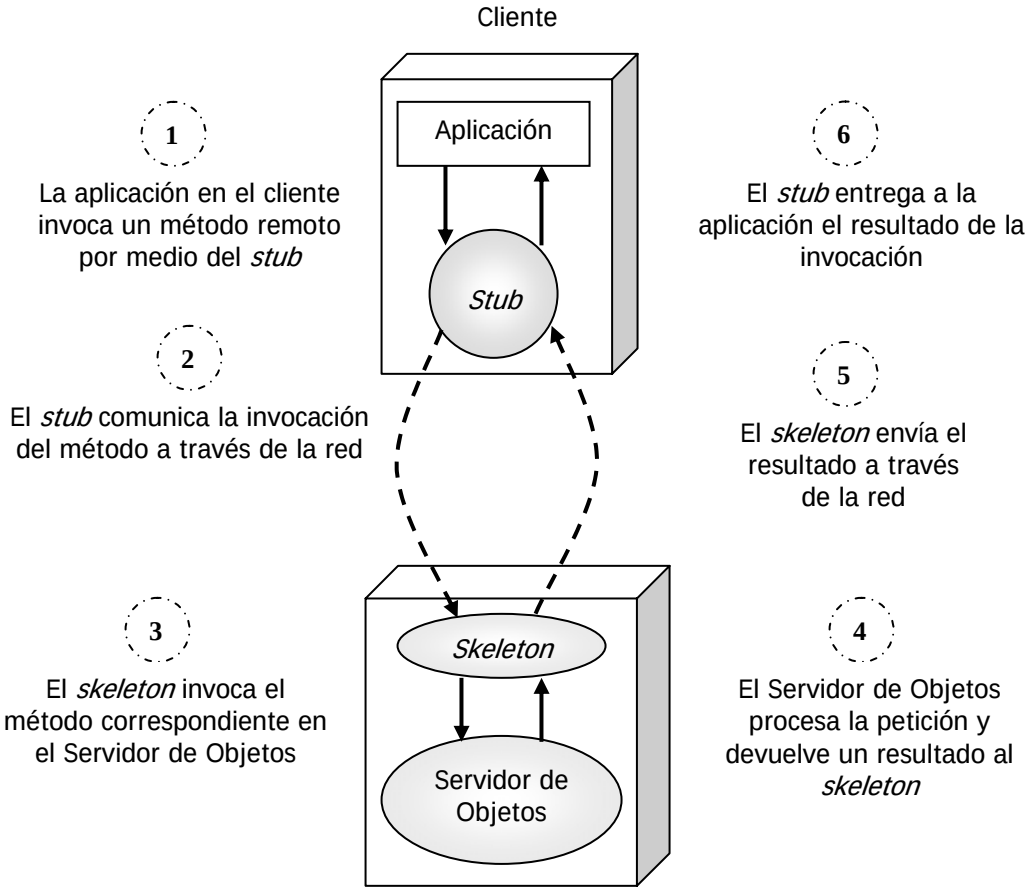



Figura 4.3. Invocación de un método remoto en el protocolo RMI.

El *stub* posee una interfaz con los mismos métodos que el servidor de objetos, pero sus

métodos no contienen lógica de negocio. En su lugar, los métodos del *stub* son utilizados para hacer peticiones al servidor de objetos y obtener resultados. Cuando un cliente invoca un método de un *stub*, la petición es comunicada a través de la red junto con los valores enviados como argumentos. El *skeleton* recibe dicha información, la analiza para descubrir qué método ha sido solicitado y lo invoca en el servidor de objetos. Cualquier valor de retorno devuelto por el método invocado es recibido por el *skeleton* y luego enviado de regreso al *stub*. Éste entrega el valor a la aplicación en el cliente como si se hubiera procesado de manera local.

### 4. 2. El Sistema De Objetos Distribuidos de Java

El acrónimo RMI (Invocación Remota de Métodos) no es específico de la tecnología Java; ya era utilizado mucho antes de que Java surgiera. En la sección anterior se ha utilizado el término RMI para describir los protocolos de objetos distribuidos de manera general. Java RMI es la versión del lenguaje Java de dicho protocolo.

La Invocación Remota de Métodos de Java (Java RMI) es un modelo de objetos distribuidos diseñado específicamente para el lenguaje Java, por lo que mantiene la semántica del modelo de objetos locales del lenguaje, facilitando de esta manera la implantación y el uso de objetos distribuidos. Java RMI ofrece algunos de los elementos críticos en cuanto a sistemas de objetos distribuidos, además de otras características particulares basadas en el hecho de que Java RMI es un sistema basado únicamente en el lenguaje Java. Tal tecnología provee facilidades de comunicación entre objetos análogas a las ofrecidas por sistemas como CORBA, y su sistema de serialización de objetos provee un medio de transferencia y solicitud de objetos entre diferentes procesos.

Java RMI permite que un objeto corriendo en una Máquina Virtual (*Virtual machine*) invoque métodos sobre un objeto remoto. Entenderemos un objeto remoto como aquél cuyos métodos pueden ser invocados por objetos que se encuentran en una MV diferente, y la invocación remota de un método como la acción de invocar un método sobre un objeto remoto.

Una de las principales metas de Java RMI consiste en facilitar el desarrollo de programas distribuidos escritos en el lenguaje Java con la misma sintaxis y la misma semántica utilizada en programas no distribuidos. Así, la invocación de un método remoto en Java tiene exactamente la misma sintaxis que la invocación de un método local.

El empleo de objetos distribuidos de Java implica varias ventajas como la orientación a

objetos misma, la movilidad de las aplicaciones Java, los patrones de diseño, la seguridad y la recolección de basura distribuida, entre otras.

La principal desventaja de los objetos distribuidos de Java, con respecto a las llamadas a procedimientos remotos, es el rendimiento. Esta desventaja es análoga a la del mismo lenguaje Java por ser un lenguaje interpretado, con respecto a los lenguajes completamente compilados como C pero, de la misma manera, todas las características positivas del modelo de objetos distribuidos de Java lo hacen una alternativa bastante interesante con respecto a sus contrapartes procedurales de más bajo nivel.

### 4. 2. 1. Metas del Sistema Java RMI

Las metas que se pretenden alcanzar al soportar objetos distribuidos en Java son:

- Proporcionar invocación remota de objetos que se encuentran en máquinas virtuales diferentes.
- Integrar el modelo de objetos distribuidos en el lenguaje Java de una manera natural, conservando en la medida de lo posible la semántica de los objetos Java.
- Hacer tan simple como sea posible la escritura de aplicaciones distribuidas.

### 4. 2. 2. Coincidencias y diferencias entre el modelo de objetos local y distribuido de Java

Como mencionamos, el modelo de objetos distribuidos de Java se desarrolló teniendo como meta acercarlo lo más posible a su modelo de objetos locales. Como es de esperar, un objeto remoto no puede ser exactamente igual a uno local, pero es similar en dos aspectos muy importantes:

- Se puede pasar una referencia a un objeto remoto, como argumento o como valor de retorno en la invocación de cualquier método, ya sea local o remoto.
- Se puede forzar una conversión de tipos de un objeto remoto a cualquier interfaz remota, mediante la sintaxis normal de Java que existe para este propósito.

Sin embargo el modelo de objetos distribuidos difiere con el modelo de objetos locales en los siguientes aspectos:

- Los clientes de los objetos remotos interactúan con las interfaces remotas y nunca con las clases que implementan dichas interfaces.
- Los argumentos de los métodos remotos, así como los valores de retorno, son pasados por copia y no por referencia.
- Los clientes deben tener en cuenta excepciones adicionales referentes a la invocación remota de los métodos.

### 4. 2. 3. Un Panorama General de Java RMI

#### El modelo de Capas

El Sistema Java RMI se puede analizar en términos de un modelo de capas, cada una con una función específica. El programador sólo debe ocuparse de la capa de Aplicación ya que las demás capas son responsabilidad del sistema Java RMI.

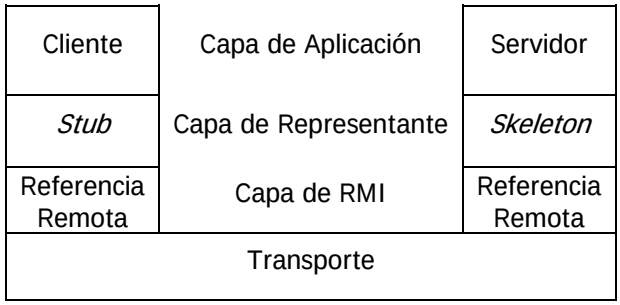



Figura 4. 4. Modelo de capas de Java RMI.

- Capa de Aplicación

En esta capa se encuentran los objetos que implementan a la aplicación. En general en este nivel se distinguen dos tipos de agentes: los clientes, que son objetos que invocan métodos o hacen peticiones a otros objetos remotos y los servidores, que son objetos que reciben peticiones y las procesan. En Java RMI se implementa un registro (*RMIregistry*) que actúa como servidor de registro y permite la localización de objetos remotos.

Las funciones que realizan un cliente y un servidor se listan en la figura 4.5:

| Funciones del Servidor | Funciones del Cliente |
|---|---|
| Crear un objeto remoto.<br>Crear una referencia al objeto remoto y hacerla disponible a los posibles clientes.<br>Esperar a que un cliente invoque un método del objeto remoto. | Obtener una referencia a un objeto remoto en el servidor.<br>Invocar un método remoto. |

Figura 4.5. Funciones de un programa cliente y un programa servidor en Java RMI.

- Capa de Representantes

En esta capa se encuentran los objetos que actúan como representantes locales de objetos remotos. En Java RMI existen dos tipos de representantes: los *stubs* del lado de los clientes y los *skeletons* del lado de los servidores.

En Java RMI un *stub* de un objeto remoto implementa el mismo conjunto de interfaces remotas que el objeto remoto al cual representa.

Cuando se invoca algún método de un *stub*, éste realiza las siguientes acciones:

- Inicia una conexión con la máquina virtual que contiene al objeto remoto.
- Transmite los parámetros de la invocación a la máquina virtual remota.
- Espera por el resultado de la invocación.
- Recibe el valor de retorno o la excepción.
- Devuelve el valor a quien lo llamó.

Cuando un *skeleton* recibe una invocación, realiza las siguientes acciones:

- Recibe los parámetros necesarios enviados por el cliente para la ejecución del método remoto.
- Invoca el método sobre el objeto remoto implementado.
- Envía los resultados de vuelta al cliente.

Tanto los *stubs* como los *skeletons* en Java, son generados por el compilador de Java RMI, llamado *rmic*.

- Capa de Referencias Remotas

En esta capa se realiza la interpretación de las referencias a objetos remotos, las referencias locales a *stubs* o *skeletons* se resuelven utilizando sus contrapartes remotas, es decir, para el nodo cliente el *stub* es la representación del objeto que está en el servidor y para el servidor el *skeleton* es la representación del objeto invocado desde el cliente. Todos los datos necesarios en la invocación de un método remoto se transfieren en la capa de transporte.

- Capa de Transporte

En esta capa se encuentra el protocolo de comunicación. Se encarga de transportar los mensajes que intercambian los distintos objetos. Java RMI utiliza por omisión el protocolo TCP/IP.

### Interfaces y Clases en Java RMI

Los paquetes de Java que constituyen el sistema Java RMI son los siguientes:

**java.rmi**. Contiene Clases, Interfaces y Excepciones vistas por los clientes.
**java.rmi.server**. Contiene Clases, Interfaces y Excepciones vistas por los servidores.
**java.rmi.registry**. Contiene Clases, Interfaces y Excepciones útiles para localizar y registrar objetos remotos.
**java.rmi.dgc**. Contiene Clases, Interfaces y Excepciones para la recolección de basura de manera distribuida.
**java.rmi.activation**. Contiene Clases, Interfaces y Excepciones para la activación de objetos remotos.

## 4. 3. Utilización de Java RMI

Un sistema Java RMI está compuesto por los siguientes elementos:

- Definiciones de Interfaces para servicios remotos.
- Implementaciones de dichas interfaces.
- Archivos *stub* y *skeleton*.
- Un servidor que provea servicios remotos.
- Un sistema de registro que permita a los clientes localizar a los objetos remotos.
- Un programa cliente que necesite los servicios remotos.

A continuación, construiremos paso a paso un sistema Java RMI muy sencillo y, para simplificar más las cosas, utilizaremos un mismo directorio para los códigos del cliente y del servidor.

Asumiendo que el sistema Java RMI ya ha sido diseñado, realizaremos los siguientes pasos para construirlo:

1. Escribir y compilar el código Java de las interfaces.
2. Escribir y compilar el código Java de las clases que implementan las interfaces.
3. Generar los archivos *stub* y *skeleton* a partir de las clases de implementación.
4. Escribir el código Java de un programa que provea servicios remotos (servidor).
5. Escribir el código Java de un programa que necesite los servicios remotos (cliente).
6. Instalar y correr el sistema.

- **Primer paso: La interfaz.**

El primer paso consiste en escribir y compilar código java para la interfaz de un servicio remoto. La interfaz Calculadora define los siguientes métodos remotos:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Calculadora extends Remote {
    public long adicion(long a, long b) throws RemoteException;
    public long sustraccion(long a, long b) throws RemoteException;
    public long producto(long a, long b) throws RemoteException;
    public long cociente(long a, long b) throws RemoteException;
}
```

Debemos notar que esta interfaz extiende a la clase Remote, y que cada uno de sus métodos declara el posible lanzamiento de una excepción RemoteException.

Ya escrito el código, procedemos a compilar la interfaz recién creada en el directorio destinado a nuestro sistema de ejemplo. Desde la línea de comandos tecleamos:

```
javac Calculadora.java
```

- **Segundo paso: La implementación.**

Ahora escribiremos la implementación del servicio remoto. Ésta es la clase CalculadoraImpl:

```
import java.rmi.server.UnicastRemoteObject;
import java.rmi.RemoteException;

public class CalculadoraImpl extends UnicastRemoteObject
   implements Calculadora {

   public CalculadoraImpl() throws RemoteException {
      super();
   }

   public long adicion(long a, long b) throws RemoteException {
      return a + b;
   }

   public long sustraccion(long a, long b) throws RemoteException{
      return a - b;
   }

   public long producto(long a, long b) throws RemoteException {
      return a * b;
   }

   public long cociente(long a, long b) throws RemoteException {
      return a / b;
   }
}
```

Podemos notar que nuestra clase de implementación importa el paquete java.rmi.server para utilizar la clase UnicastRemoteObject y así formar parte del sistema RMI. Cuando una clase extiende de UnicastRemoteObject debe proveer un constructor que declare el posible lanzamiento de una RemoteException. Cuando este constructor invoca el método super(), se activa el código de UnicastRemoteObject que realiza la integración a Java RMI y la inicialización del objeto remoto.

Una vez escrita la implementación de nuestra interfaz debemos compilarla en el mismo directorio:

```
javac CalculadoraImpl.java
```

- **Tercer paso: *Stub* y *skeleton*.**

Una vez compilada la implementación de nuestra interfaz, debemos utilizar el compilador de Java RMI, llamado *rmic*, para generar los archivos *stub* y *skeleton*. Este compilador corre sobre el archivo de *bytecodes* que implementa nuestro servicio remoto, es decir, CalculadoraImpl.class:

*rmic* CalculadoraImpl

Después de haber compilado la clase CalculadoraImpl con *rmic* se debe encontrar el archivo Calculadora_Stub.class y, si se está corriendo el Java 2 SDK, el archivo Calculadora_Skel.class.

- **Cuarto paso: El programa servidor.**

Los servicios remotos de RMI deben encontrarse en un proceso servidor. La clase ServidorCalculadora es un servidor muy simple que cuenta con lo estrictamente esencial para brindar servicios.

```
import java.rmi.Naming;

public class ServidorCalculadora {

  public ServidorCalculadora () {

    String nombre = "ServicioCalculadora";
    try {
      Calculadora c = new CalculadoraImpl();
      Naming.rebind("rmi://localhost:1099/" + nombre, c);
       } catch (Exception e) {
      System.out.println("Excepción: " + e);
    }
  }

  public static void main(String args[]) {
    new ServidorCalculadora();
  }
}
```

Antes de que un cliente pueda invocar el método de un objeto remoto proporcionado a través del servicio que brinda nuestro servidor, el cliente debe obtener una referencia a dicho objeto remoto. Esto puede realizarse obteniéndola como el valor de retorno de un método que la proporcione o como parte de una estructura de datos que la contenga.

Para este fin, Java RMI provee un objeto remoto particular: el registro RMI (*RMI registry*). Éste permite a los clientes localizar referencias a objetos remotos en el servidor. El registro RMI es un servicio de nombrado para objetos remotos que permite la localización de éstos por medio de su nombre.

La interfaz java.rmi.Naming es utilizada como una herramienta para registrar y localizar objetos remotos en el servicio de registro. Una vez que un objeto remoto ha sido registrado en el servidor a través del registro RMI, los clientes en cualquier otro nodo pueden localizar dicho objeto remoto por su nombre, obtener una referencia a él e invocar sus métodos. El registro puede compartirse por todos los procesos servidores que se ejecuten en un nodo o cada uno de ellos puede crear y utilizar su propio registro.

La clase ServidorCalculadora crea un nombre para el objeto remoto a través de la sentencia rmi://localhost:1099/ServicioCalculadora;

Esta sentencia incluye el nombre del nodo o host en el que se ejecutan el registro RMI y el objeto remoto, y un nombre, ServicioCalculadora, que identifica al objeto remoto en el registro. El registro RMI se ejecuta en cada una de las máquinas que provean objetos remotos y por defecto escucha las peticiones de los clientes por el puerto 1099.

El código necesita entonces agregar el nombre al registro RMI que corre en el nodo servidor. Esto se hace en la sentencia

```
Naming.rebind(nombre, c);
```

Al llamar al método rebind() se hace una llamada al registro RMI que se encuentra en el propio nodo (*localhost*). Esta llamada puede generar una RemoteException que debe ser cachada por lo que la clase ServidorCalculadora maneja la excepción dentro del bloque try / catch.

Compilamos el código del programa servidor utilizando el compilador de Java:

```
javac ServidorCalculadora.java
```

- **Quinto paso: El programa cliente.**

El código para el programa cliente se muestra a continuación:

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;

public class ClienteCalculadora {

    public static void main(String[] args) {
        try {
Calculadora calc = (Calculadora);
String nombre = "ServicioCalculadora";
Naming.lookup("rmi://remotehost/"+ nombre);
            System.out.println(calc.sustraccion(4, 3) );
            System.out.println(calc.adicion(4, 5) );
            System.out.println(calc.producto(3, 6) );
            System.out.println(calc.cociente(9, 3) );
        }
        catch (MalformedURLException murle) {
            System.out.println();
            System.out.println("MalformedURLException");
            System.out.println(murle);
        }
        catch (RemoteException re) {
            System.out.println();
            System.out.println("RemoteException");
            System.out.println(re);
        }
        catch (NotBoundException nbe) {
            System.out.println();
            System.out.println("NotBoundException");
            System.out.println(nbe);
        }
        catch (java.lang.ArithmeticException ae) {
            System.out.println();
            System.out.println("java.lang.ArithmeticException");
            System.out.println(ae);
        }
    }
}
```

El programa cliente construye un nombre para localizar al objeto remoto Calculadora. El método Naming.lookup es utilizado por el cliente para localizar al objeto remoto por su nombre en el registro RMI del nodo servidor. Cuando tal método es ejecutado, se crea una URL que especifica el nombre del nodo donde se encuentra el programa servidor (*remotehost*). El nombre utilizado como argumento del método Naming.lookup presenta la misma sintaxis que el nombre utilizado en el método Naming.rebind del servidor, que se explicó anteriormente.

Después, el cliente crea un objeto Calculadora e invoca los métodos remotos del objeto.

Compilamos el código del programa cliente utilizando el compilador de Java:

```
javac ClienteCalculadora.java
```

- **Sexto paso: Puesta en marcha.**

Para correr el sistema es necesario abrir tres pantallas de consola: una para el servidor, otra para el cliente y otra para el registro RMI.

Comenzaremos con el servicio de registro. Debemos estar ubicados en el directorio que contiene las clases que hemos escrito y, una vez ahí, tecleamos lo siguiente:

```
rmiregistry
```

El registro comenzará a correr y podremos pasar a la siguiente consola. De manera alternativa, si deseamos que el servicio de registro se ejecute en segundo plano, debemos teclear rmiregistry &. Así podemos utilizar la misma consola para ejecutar nuestro programa servidor.

Ahora arrancaremos el servidor ServidorCalculadora tecleando lo siguiente:

```
java ServidorCalculadora
```

El servidor comenzará a ejecutarse, cargará en el servicio de registro la implementación del servicio que provee y esperará a que un cliente se conecte.

En la última consola arrancaremos el programa cliente:

```
java ClienteCalculadora
```

Se podrá apreciar la siguiente salida en pantalla:

```
1
9
18
3
```

Hemos creado un sistema Java RMI funcional y, aunque hemos utilizado diferentes consolas en una misma computadora, el sistema Java RMI realiza su ejecución como si se tratara de máquinas virtuales diferentes.

# CAPÍTULO 5

# Transacciones distribuidas.

Un sistema de información distribuido permite que los usuarios almacenen, accedan y modifiquen datos de manera transparente desde o hacia diversas computadoras, manteniendo la integridad de los datos durante alguna falla. Dada la naturaleza de una consulta, de lectura o actualización, a veces simplemente no se puede reactivar su ejecución, puesto que algunos datos pueden haber sido modificados antes de un error. El no tomar en cuenta estos factores puede propiciar que la información en las bases de datos del sistema sea incoherente.

Un sistema transaccional es aquel que asegura que una unidad de trabajo se realice completamente o que no tenga efecto. El manejo de transacciones libera al programador de la responsabilidad de ocuparse del acceso a datos, incluyendo modificaciones concurrentes en los datos, tolerancia a fallas y programación multiusuario.

## 5. 1. Transacciones.

Una transacción es un grupo de sentencias que representan una unidad de trabajo y deben ejecutarse en su totalidad. Las transacciones son secuencias de operaciones – lecturas, escrituras o actualizaciones - que llevan a un sistema de un estado consistente a otro. Se dice que una base de datos se encuentra en un estado consistente si obedece a todas las restricciones de integridad definidas sobre ella.

Por ejemplo, si deseáramos implementar una transferencia de fondos entre dos cuentas bancarias de un mismo cliente, podríamos realizar dos operaciones:

- Realizar un retiro de la cuenta A
- Realizar un depósito en la cuenta B

Si la primera operación es ejecutada entonces la segunda operación debe ejecutarse correctamente, y viceversa. Si sólo la primer operación se realizara, el monto retirado de la cuenta A nunca se habría depositado en la cuenta B, afectando negativamente el saldo la cuenta A de nuestro cliente. De manera contraria, si sólo se realizara la segunda operación, nuestro cliente se

vería beneficiado al ver incrementado su saldo en la cuenta B, pero afectando a la institución bancaria encargada de realizar dicha transferencia, pues el monto depositado no provendría del cliente que solicitó la operación.

En este caso, es necesario implementar una transacción que involucre las operaciones de retiro y depósito para asegurar que ambas se ejecuten en su totalidad o que ninguna de las dos surta efecto.

- *Commit*

Una transacción siempre termina, aún en la presencia de fallas. Si ninguna de las operaciones englobadas por una transacción presenta una falla se dice que la transacción es exitosa. En este caso, la transacción hace efectivas las modificaciones realizadas por las operaciones que involucra, llevando el sistema a un nuevo estado consistente. A este proceso se le conoce como hacer un *commit*.

- *Rollback*

Si existe por lo menos una falla dentro de las operaciones de una transacción, se dice que ésta aborta. Su ejecución es detenida y todas las operaciones ejecutadas hasta el momento del error son deshechas, regresando la base de datos al estado consistente en que se encontraba antes de iniciar la transacción. A esta operación se le conoce como hacer un *rollback*.

### 5. 1. 1. Propiedades de una transacción

Para garantizar que los datos involucrados en una transacción sean compartidos de manera segura y confiable, ésta debe contar con cuatro propiedades básicas: atomicidad, consistencia, aislamiento y durabilidad. Dichas propiedades son conocidas por el acrónimo *ACID* (*Atomicity, Consistency, Isolation, Durability*), de acuerdo con sus siglas en inglés.

- Atomicidad (*Atomicity*)

Significa que la transacción se considera completa si, y sólo si, todas las operaciones que involucra son ejecutadas exitosamente. Si cualquiera de las operaciones en la transacción falla, la transacción debe abortar.

Por ejemplo, si deseáramos registrar la orden de compra de un cliente en el sistema ventas de alguna empresa, la orden de compra y la información asociada a la tarjeta de crédito del comprador deben utilizarse para retirar el monto necesario de su cuenta bancaria. Si por alguna razón no pudiera registrarse la orden de compra (por ejemplo, el artículo se encuentra agotado), no debería haber ningún recargo sobre la cuenta del cliente. De manera similar, si no pudiera realizarse el retiro de la cuenta (porque el comprador ha excedido el límite de su crédito), la orden de compra no debería registrarse en el sistema. En aplicaciones grandes, pueden existir requerimientos de atomicidad mucho más complejos, donde existe una codependencia entre la ejecución de diversas operaciones.

- Consistencia (*Consistency*)

Significa que una transacción debe llevar los datos de un estado consistente a otro, preservando la semántica de éstos y su integridad referencial.

El registro de una orden de compra sin el correspondiente recargo crearía un estado inconsistente en el sistema de nuestro ejemplo. Se suscitaría un grave problema si el responsable del sistema de ventas se percatara de que una orden de compra ha sido registrada sin haberle cobrado al comprador. De esta manera, la consistencia de un sistema es un requerimiento clave para las aplicaciones transaccionales.

- Aislamiento (*Isolation*)

Significa que cualquier cambio hecho en los datos por una transacción inicial es invisible para cualquier otra transacción, mientras la primera transacción no haya terminado. El aislamiento garantiza que varias transacciones concurrentes produzcan los mismos resultados en los datos y que una transacción no acceda a datos que están siendo modificados en forma concurrente.

Una consulta en nuestro sistema de ventas no debería reportar una orden de compra que todavía se encuentra en proceso de realización hasta que la transacción que involucra dicha orden haya hecho un *commit*. El aislamiento es importante para brindar una vista consistente del sistema a otras aplicaciones. De esta manera, los cambios realizados sólo deben percibirse cuando su ejecución haya sido completada.

- Durabilidad (*Durability*)

Significa que los resultados de transacciones finalizadas deben ser permanentes y no pueden ser borrados de la base de datos debido a fallas en el sistema. Los errores que ocurran después de un *commit* no deben causar la pérdida de datos.

En nuestro ejemplo, los datos de una orden registrada exitosamente no deben perderse por fallas posteriores en la aplicación.

### 5. 1 . 2. Tipos de transacciones

Las transacciones pueden clasificarse de varias maneras de acuerdo con los algoritmos y las técnicas que se utilicen para manejarlas. De manera general, una transacción puede clasificarse dentro de las siguientes dimensiones:

- **Por la distribución de los datos**. Si las operaciones comprendidas dentro de una transacción se efectúan sobre una sola base de datos, o sobre bases de datos diferentes pero comprendidas dentro del mismo servidor y un mismo manejador, se dice que se trata de una transacción local. Si las operaciones que comprende una transacción se realizan sobre diferentes bases de datos, distribuidas en una red de comunicaciones, se dice que la transacción es distribuida.

- **Por su durabilidad.** Dado que los resultados de una transacción que realiza un *commit* son durables, la única forma de deshacer sus efectos es utilizando otra transacción que los modifique. A este tipo de transacciones se les conoce como transacciones compensatorias.

- **Por las características de las bases de datos.** Si las operaciones de una transacción se ejecutan sobre bases de datos homogéneas, es decir, que cuentan con las mismas características y con el mismo manejador de base de datos, se dice que la transacción es homogénea. En caso contrario, se dice que la transacción es de tipo heterogéneo.

- **Por su duración.** Tomando en cuenta el tiempo que transcurre desde que se inicia una transacción hasta que se realiza un *commit* o un *rollback*, las transacciones pueden ser de corta y larga vida. Las transacciones de vida corta (también conocidas como

transacciones en línea) se caracterizan por tiempos de respuesta muy cortos y por acceder a una porción de datos relativamente pequeña. Por otro lado, las transacciones de vida larga (o de tipo *batch*) toman tiempos relativamente largos y acceden a grandes porciones de datos.

- **De acuerdo con su estructura.** Considerando la estructura que puede tener una transacción se examinan dos aspectos. Si ninguna de las operaciones contenidas dentro de una transacción es otra transacción, se dice que la transacción es de tipo plano. Si una transacción puede contener a su vez subtransacciones se dice que las transacciones están anidadas.

## 5. 2. Transacciones Distribuidas.

Los sistemas distribuidos frecuentemente necesitan acceder y actualizar múltiples recursos transaccionales para lograr una cierta meta global. Considere, por ejemplo, una aplicación en una agencia de viajes. Crear un itinerario para un viaje de negocios con un boleto confirmado y pagado requiere la terminación acertada de la autenticación del usuario, del procesamiento de la tarjeta de crédito, y de la reservación del vuelo, así como de la creación misma del itinerario. Cada una de estas operaciones involucra sistemas transaccionales de cooperación independientes. A la operación conjunta de todas estas operaciones se le conoce como transacción distribuida.

Las transacciones distribuidas son más complejas que las transacciones no distribuidas debido al estado latente de que alguno de los manejadores falle y con ello el sistema global no opere de manera correcta. En una red, una transacción fallida puede ser difícil de distinguir de una que es simplemente lenta. Los manejadores locales de las bases de datos involucrados en una transacción no conocen el estado en que se encuentra el resto de los manejadores con los que cooperan y, por lo tanto, no pueden coordinar sus propias transacciones de manera independiente.

Un manejador de base de datos es un sistema que brinda servicios a diferentes aplicaciones para que éstas puedan acceder a una base de datos. El manejador de base de datos brinda y hace cumplir las propiedades *ACID* para operaciones que sobre él se ejecuten. Algunos de estos manejadores podrían proveer acceso a una base de datos relacional, brindando soporte de almacenamiento de datos a una cierta aplicación.

La solución más común al problema de coordinar transacciones distribuidas es introducir un nuevo participante, llamado manejador de transacciones distribuidas. Éste actúa como mediador entre los diferentes manejadores de bases de datos locales involucrados. En la figura 5.1 se puede observar a tres participantes en una transacción distribuida: la aplicación transaccional, el manejador de transacciones distribuidas, que coordina las transacciones de los múltiples manejadores de bases de datos, y los propios manejadores locales, dándole a la aplicación transacciones *ACID* por medio de recursos múltiples.

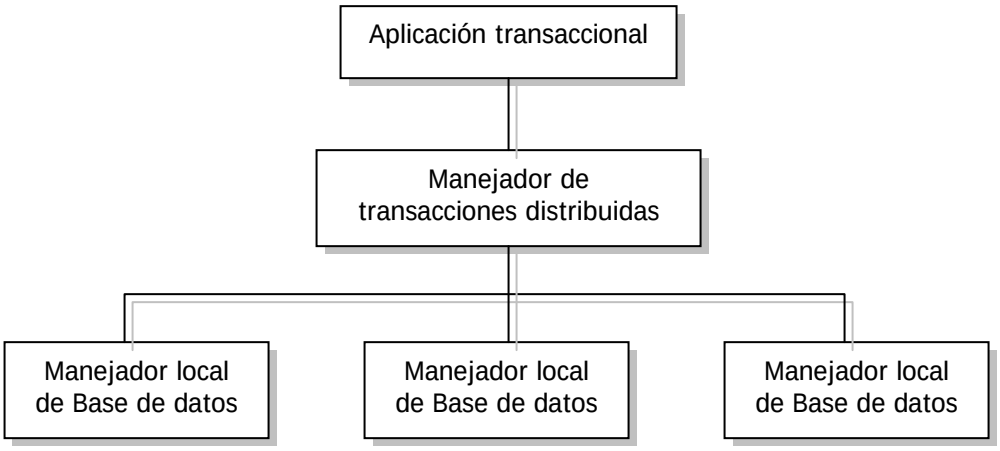



Figura 5.1. Participantes dentro de una transacción distribuida

En cualquier momento de una transacción distribuida, el manejador de transacciones mantiene una asociación entre las transacciones que se ejecutan en los manejadores locales (cada una de las cuales tiene un identificador único global), los hilos de la aplicación y las conexiones de los manejadores a las bases de datos.

### 5. 2. 1. El *commit* de dos Fases

Dado que los diferentes manejadores de bases de datos locales en una transacción distribuida no están conscientes del resto de los manejadores, no pueden cooperar directamente con ellos. Por esto, el manejador de transacciones distribuidas le indica a cada manejador cuándo hacer un *commit* o un *rollback*, de acuerdo con el estado global de la transacción. La coordinación de todos los manejadores se realiza a través de un protocolo conocido como *commit* de dos fases.

El *commit* de dos fases (*two-phase commit*) es un protocolo muy simple y elegante que asegura la atomicidad de las transacciones distribuidas. Extiende los efectos de una operación local de *commit* a transacciones distribuidas poniendo de acuerdo a todos los nodos involucrados en la

ejecución de una transacción antes de que los cambios hechos por la transacción sean permanentes.

- Fase 1

El manejador de transacciones ordena a cada manejador de base de datos que se prepare para hacer un *commit*, esto es, que realice todas las operaciones que le han sido asignadas y que espere una nueva indicación, ya sea para realizar los cambios de manera permanente (*commit*) o deshacer sus modificaciones (*rollback*). Cada uno de los manejadores locales responde, indicando si la ejecución de sus operaciones fue exitosa o no.

- Fase 2

Si todos los manejadores de bases de datos han indicado un resultado exitoso en la ejecución de sus operaciones, el manejador de transacciones les indica que hagan un *commit* para hacer efectivos todos los cambios. Posteriormente, el manejador de transacciones le indica a la aplicación que la transacción distribuida fue exitosa. No obstante, si alguno de los manejadores reporta la ejecución fallida de sus operaciones, el manejador de transacciones le ordena a todos los manejadores involucrados que realicen un *rollback* para deshacer los cambios. Posteriormente, el manejador de transacciones le indica a la aplicación que la transacción distribuida fue fallida.

La operación del *commit* de dos fases entre un manejador de transacciones y tres manejadores de bases de datos se presenta en la figura 5.2.

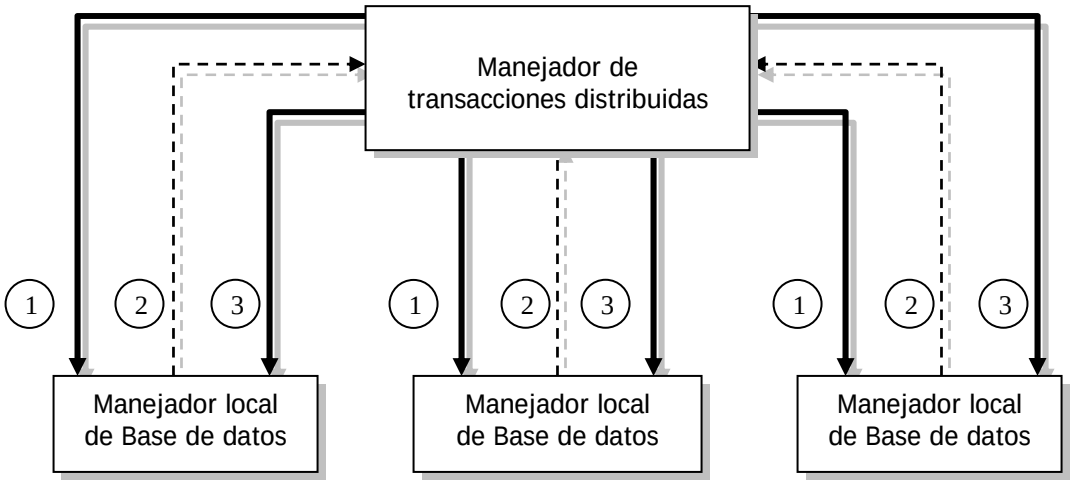



Figura 5.2. Protocolo *commit* de dos fases.

<table>
<tr><th>FASE 1</th><th>FASE 2</th></tr>
<tr><td>1. El manejador de transacciones distribuidas indica a cada uno de los manejadores de bases de datos locales que realice las operaciones que tiene asignadas.</td><td rowspan="2">3. Sin fallas en los manejadores, el manejador de transacciones distribuidas le indica a todos los manejadores que realicen un commit. Si existe una falla en algún manejador, el manejador de transacciones indica a todos los manejadores que hagan un rollback.</td></tr>
<tr><td>2. Los manejadores de bases de datos locales realizan las operaciones y avisan al manejador de transacciones si el proceso que realizaron fue exitoso.</td></tr>
</table>

Figura 5.2. (Continuación) Protocolo *commit* de dos fases.

El esquema presentado en la figura 5.2 es de tipo centralizado ya que la comunicación se realiza sólo entre el manejador de transacciones y los manejadores de bases de datos; éstos no se comunican entre sí.

### 5. 2. 2. Aspectos relacionados al procesamiento de transacciones

Los siguientes son los aspectos más importantes relacionados con el procesamiento de transacciones:

- **Modelo de estructura de transacciones.** Es importante considerar si las transacciones son planas o pueden estar anidadas.
- **Consistencia de la base de datos interna.** Se deben satisfacer siempre las restricciones de integridad de los datos cuando una transacción pretende hacer un *commit*.
- **Protocolos de confiabilidad.** En transacciones distribuidas es necesario introducir medios de comunicación entre los diferentes nodos de una red para garantizar la atomicidad y durabilidad de las transacciones. Así también, se requieren protocolos de recuperación local y para efectuar *commits* de manera global, por ejemplo, el protocolo del *commit* de dos fases.
- **Algoritmos de control de concurrencia.** Los algoritmos de control de concurrencia deben sincronizar la ejecución de transacciones concurrentes. La consistencia entre transacciones se garantiza mediante su aislamiento.
- **Protocolos de control de réplicas.** El control de réplicas se refiere a cómo garantizar la consistencia mutua de datos replicados.

# CAPÍTULO 6

# Desarrollo de un sistema de base de datos distribuida

Uno de los objetivos fundamentales de un sistema de información es contar no sólo con recursos de información, sino también con los mecanismos necesarios para poder encontrar y recuperar esos recursos. Por tal razón, la utilización de manejadores de bases de datos se ha convertido en un elemento indispensable no sólo para el funcionamiento de los grandes motores de búsqueda y la recuperación de información a lo largo y ancho de la Web, sino también para la creación de Intranets y muchos otros sistemas en los que es necesario procesar grandes volúmenes de datos. La mayoría de las veces estos sistemas son bastante complejos y es necesario utilizar herramientas que faciliten su construcción.

## 6. 1. Paradigmas en el ámbito de la Ingeniería de Software

Para la realización de un sistema se requiere de paradigmas que ayuden a los creadores de ese sistema a analizarlo y diseñarlo. Actualmente, el uso de la palabra *paradigma* tiene sus raíces en la definición dada por el científico y filósofo Thomas Kuhn en su libro *The Structure of Scientific Revolutions* en el año de 1962:

Un paradigma es "... *una constelación de conceptos, valores, percepciones y prácticas compartidas por una comunidad que forma una visión particular de la realidad y que es la base con la que una comunidad se organiza a sí misma*".

En el ámbito de la Ingeniería de Software, los principales paradigmas que predominan, y en los cuales se pensó para construir este sistema, son:

- El paradigma estructurado procedural.
- El paradigma orientado a objetos.

La primera opción, a pesar de que ha sido empleada durante mucho tiempo en proyectos realmente complejos, presenta tres grandes deficiencias:

- Los productos que resultan al emplear estas técnicas son pocos flexibles.
- Los programadores que lo utilizan tienden a concentrarse en el diseño y la puesta en práctica del sistema, sin tomar en cuenta su vida posterior (el mantenimiento).
- No alientan al programador a aprovechar el trabajo de proyectos anteriores.

La desventaja de utilizar un paradigma que se concentra en el diseño inicial del sistema se hace evidente si se toma en cuenta que la vida útil de un producto de software puede ser cinco o seis veces más grande que el lapso en que se desarrolla. La fase de mantenimiento de un sistema es tan importante que cualquier método de diseño debe tener como objetivo principal producir sistemas que faciliten su propio mantenimiento.

La reutilización de código es otro de los factores que no se toma en cuenta para los métodos de diseño tradicionales. Cualquier programador que desarrolla un sistema nuevo debe escribir una buena cantidad de código que ha escrito anteriormente una y otra vez. Las rutinas de búsqueda, ordenamiento, manejo de menús y despliegue de ventanas, entre otras, se repiten continuamente en proyectos diferentes. Los métodos de desarrollo de software deben alentar al programador a utilizar el código escrito previamente ya sea por él mismo o por otros programadores.

Dentro del paradigma orientado a objetos existen muchos métodos de modelado de sistemas, de los cuales el predominante es la Técnica de Modelado de Objetos u *OMT* (*Object Modeling Technique*) pero en 1995 surgió no un método sino un lenguaje de modelado de sistemas llamado Lenguaje Unificado de Modelado o *UML* (*Unified Modeling Language)*.

### 6. 1. 1. Lenguaje Unificado de Modelado (UML)

El Lenguaje unificado de modelado (UML) es una técnica para la especificación de sistemas en todas sus fases. Ha sido desarrollado por los más importantes autores en materia de Análisis y Diseño de Sistemas y ha sido utilizado con éxito en sistemas construidos para toda clase de industrias alrededor del mundo: hospitales, bancos, comunicaciones, aeronáutica, finanzas, etc.

UML sirve para realizar modelos que permitan:

- Visualizar cómo es un sistema o cómo queremos que sea.
- Especificar la estructura y el comportamiento de un sistema.
- Hacer una plantilla que guíe la construcción de los sistemas.
- Documentar las decisiones que se han tomado.

UML consiste de:

- Reglas de simbología que aplican a cualquier tipo de modelo hecho bajo este lenguaje, por ejemplo, el modo en que se coloca un comentario en cualquier diagrama o el modo en que se aumenta la nomenclatura existente en UML.
- Diferentes tipos de diagramas: Diagramas de casos de uso, diagramas de clase, diagramas de componentes, etc. Cada diagrama está diseñado para enfocar un aspecto en particular de un sistema.

UML soporta prácticamente todas las fases de un ciclo de desarrollo de un sistema: Recopilación de requerimientos, análisis, diseño, desarrollo, puesta en práctica o instrumentación, así como en reingeniería y en cualquier actividad de desarrollo que sea susceptible de ser modelada.

Además, los principales beneficios que UML aporta son:

- Mejores tiempos totales de desarrollo.
- Mejor calidad.
- Mejor soporte a la planeación y al control de proyectos.
- Mayor independencia del personal de desarrollo.
- Mayor soporte al cambio organizacional, comercial y tecnológico.
- Alta reutilización.
- Minimización de costos.

Por todas estas razones, se optó por utilizar UML como una herramienta que facilitara el desarrollo de este sistema. Las diversas fases en las que se aplicó son, principalmente: Análisis de requerimientos, análisis, diseño y desarrollo.

### 6. 2. Análisis de Requerimientos

Muchos manejadores de bases de datos proveen un control de transacciones locales que engloban a las bases de datos con las que son construidos pero no ofrecen la posibilidad de crear transacciones a través de una red de comunicaciones. Sin embargo, cada vez existen más manejadores que permiten la explotación distribuida de distintas bases de datos aunque, en general, son escasos y muchas veces se trata de sistemas propietarios cuya adquisición implica el pago de licencias costosas y poca flexibilidad en cuanto a la interacción con bases de datos desarrolladas por otros fabricantes.

Uno de los objetivos de la tesis presentada en estas páginas es demostrar de manera práctica el procesamiento distribuido de distintas bases de datos en un *cluster* tipo *Beowulf*, que es una arquitectura de procesamiento paralelo constituida por computadoras que se pueden conseguir fácilmente y por la utilización de software libre.

La utilización de este tipo de software es uno de los requerimientos principales en el desarrollo del proyecto que aquí se presenta, por lo que el uso de un sistema propietario que ya soportara la ejecución de transacciones distribuidas no era factible o deseable.

Por otro lado, la utilización de software no propietario brinda, de manera general, dos distintas opciones para la construcción de un sistema de base de datos distribuida.

La primera opción consiste en adquirir software libre de base de datos distribuida, lo cual implica una ventaja en cuanto a que los mecanismos de distribución de datos ya están desarrollados. Con esta alternativa, el proyecto de tesis consistiría solamente en crear una aplicación que utilizara dicho sistema. Esto simplificaría enormemente el desarrollo del proyecto ya que no sería nuestra responsabilidad encargarnos del manejo distribuido de transacciones sino que nos enfocaríamos específicamente a diseñar y desarrollar una aplicación que ejemplifique la utilización del sistema adquirido.

La segunda opción es utilizar distintos manejadores de bases de datos y un lenguaje de programación, todos no propietarios, para desarrollar con ellos un mecanismo de distribución de datos entre los distintos nodos del *cluster*. En este caso, no existe un sistema de base de datos ya desarrollado sino que éste sería construido a lo largo del proyecto. La ventaja principal de esta alternativa es que brinda una mejor comprensión del procesamiento de una base de datos distribuida ya que, básicamente, todo sería analizado, diseñado y construido desde cero,

implementando nosotros mismos la distribución de datos de manera paralela y distribuida y un manejo de transacciones globales que brinde un cierto grado de integridad en las bases de datos involucradas. Evidentemente, esta opción implica un reto mayor que la anterior, donde ya se parte de un sistema que, en teoría, debería encargarse de todos estos aspectos.

Teniendo en cuenta estas dos posibilidades se investigó qué software libre podría utilizarse en cada una de ellas, con lo que se obtuvo la siguiente información:

### 6. 2. 1. Sistemas de base de datos distribuida de software libre

- **EXODUS Project Software**

El sistema manejador de almacenamiento EXODUS es un sistema cliente-servidor para el almacenamiento de objetos y permite almacenar datos, versiones de objetos, archivos que relacionan diferentes almacenes de objetos e índices que soportan un acceso eficiente a dichos objetos. Un objeto de almacenamiento es un contenedor de *bytes* que no están interpretados y puede almacenar desde unos pocos *bytes* hasta *megabytes.* El sistema manejador permite rutinas de lectura, reescritura, y de crecimiento eficiente o de reducción de objetos. Además de proveer transacciones, bloqueos basados en el control de concurrencia, y registro de recuperación. GNU E es un lenguaje de programación orientado a objetos, que extiende de C++, y desarrollado como parte del proyecto EXODUS.

- **SIOD (Scheme In One Defun/Day)**

Es un proyecto interpretado con procedimientos construidos usando Oracle Call Interfase (OCI) y servicios de SQL de Digital RDB. Puede usarse simplemente como una base de datos flexible para cargar y descargar con rapidez archivos de datos planos para su almacenamiento; o también puede ser usado para aplicar las técnicas clásicas de Manipulación Simbólica o de Inteligencia Artificial en conjuntos de datos (datasets). El principal programa puede ser orientado a *batch*, terminales o Sybase usando la librería CT en una interfaz Windows/GUI. El sistema también permite lenguaje s*cript* del ambiente *UNIX.*

- **ONYX**

Es un lenguaje basado en un modelo de vista-controlador. Onyx 4gl conecta un manejador de transacciones basado en un generador *OO-Parser* (*Parser* orientado a objetos) por medio de un *socket*. Mientras la meta del diseño de un protocolo fue mantenerlo tan simple como fuera posible, es un buen inicio para la escritura de aplicaciones comerciales de bases de datos independientes.

- **SHORE**

Es un repositorio de objetos, escalable y con alto desempeño. Es un proyecto para diseñar, implementar y evaluar un sistema de objetos persistentes que darán servicio a las necesidades de una amplia variedad de aplicaciones, incluyendo hardware y software se sistemas CAD, lenguajes de programación persistentes, sistemas de información geográfica, repositorios de datos satelitales y aplicaciones multimedia. SHORE provee un espacio de nombres jerárquico para el nombrado de objetos, y una interfaz, Unix, compatible con un campo de texto. SHORE soporta múltiples lenguajes de programación y tipos de objetos. Está diseñado para manipular información hasta *terabytes*, pero es un proyecto aún en desarrollo.

- **YOODA (Yet another Object Oriented Database)**

Es una pequeña y simple Base de datos orientada a objetos. Basado en el lenguaje de programación C++, provee el manejo de transacciones sencillas en una arquitectura multi-clientes/multi-servidores. Sus principales características son:

- Una base de datos distribuida con múltiples clientes y múltiples servidores.
- Usa CORBA como interfaz de comunicación.
- Cuenta con una interfaz para C++, la cual es usada de precompilador.
- Al programar puede generar código realmente pequeños (menor a 15000 líneas).
- Cuenta con un buen desempeño y manejo de objetos de gran tamaño.
- El acceso distribuido es completamente transparente, el usuario únicamente indica el servidor en el cual se creará el objeto.
- Emplea el mecanismo c*ommit de dos fases* para el manejo distribuido de transacciones.

- **GOODS**

Es un sistema manejador de base de datos distribuida orientado completamente a objetos, usando un modelo de cliente activo. Brinda un alto desempeño al usar múltiples hilos en la base de datos del servidor, es independiente del lenguaje y la aplicación que lo utilice. La interfaz de la aplicación del cliente está construida por medio del protocolo de meta-objetos y proporcionando transparencia en la interacción con otros lenguajes de programación, permitiendo definir varias estrategias de acceso a las bases de datos para una aplicación específica y separarlos del código de la aplicación.

- **MARIPOSA**

Mariposa es un proyecto de investigación de la universidad de California en Berkeley. Resuelve problemas fundamentales del manejo de datos distribuidos. Permite que distintos sistemas manejadores de base de datos separados y bajo diversos dominios administrativos trabajen juntos para procesar peticiones de búsquedas a las bases de datos. Ofrece los siguientes beneficios:

- **Escalabilidad**. Adaptabilidad a un número grande de sitios. En un ambiente WAN, puede haber un número grande de sitios. El objetivo del proyecto es Mariposa es escalar a 10,000 servidores.
- **Autonomía local**. Cada sitio debe tener el control de sus recursos. Esto incluye cada objeto almacenado y cada búsqueda que se ejecute. La búsqueda y la asignación de datos no se realizan por un nodo central o autoritario.
- **Movilidad de los datos**. Promete ser fácil y eficiente para cambiar la ubicación de un objeto. Preferentemente, el objeto debería permanecer disponible durante el movimiento.
- **Ninguna sincronización global**. Las actualizaciones y cambios de esquema no fuerzan a un sitio a sincronizar con el resto.
- **Política fácilmente configurable.** Provee facilidades para que un administrador de base de datos local pueda modificar el comportamiento de un sitio en Mariposa. Un sistema Mariposa debería responder de manera amable a cambios de la actividad de usuario y el modelo de acceso de datos para mantener el tiempo de respuesta bajo y el alto rendimiento del sistema.

### 6. 2. 2. Software libre para el desarrollo de un sistema de base de datos distribuida

- **El lenguaje de programación Java**

Debido a que en nuestros estudios de licenciatura y en nuestro ejercicio profesional habíamos estudiado y utilizado el lenguaje de programación Java, decidimos evaluar sus características para el desarrollo de esta tesis. A continuación mencionamos las más importantes:

- **Orientado a objetos.** Java fue diseñado como un lenguaje orientado a objetos desde el principio. Los objetos agrupan en estructuras encapsuladas tanto sus datos como los métodos que manipulan esos datos. La tendencia del futuro, a la que Java se suma, apunta hacia la programación orientada a objetos, especialmente en entornos cada vez más complejos y basados en red.
- **Interpretado y compilado a la vez.** Java es compilado ya que su código fuente se transforma en una especie de código máquina llamado *bytecode*, semejante a instrucciones en lenguaje ensamblador. Por otra parte, es interpretado, ya que los *bytecodes* se pueden ejecutar directamente sobre cualquier máquina que cuente con un intérprete, independientemente del sistema operativo en que se ejecute.
- **Robusto.** Java fue diseñado para crear software altamente confiable. Para ello proporciona numerosas comprobaciones en compilación y en tiempo de ejecución. Además facilita el acceso a memoria pues exime a los programadores de errores cometidos en el manejo de punteros y su esquema de recolección de basura elimina la necesidad de liberación explícita de memoria.
- **Lenguaje multiplataforma.** Java está diseñado para soportar aplicaciones que serán ejecutadas en los más variados entornos de red, desde Unix hasta Windows, pasando por Mac y estaciones de trabajo, sobre arquitecturas distintas y con sistemas operativos diversos. Para soportar estos requisitos de ejecución tan variados, el compilador de Java genera *bytecodes*: un formato intermedio indiferente a la arquitectura, diseñado para transportar el código eficientemente a múltiples plataformas hardware y software.
- **Portable.** La indiferencia a la arquitectura representa sólo una parte de su portabilidad. Además, Java especifica los tamaños de sus tipos de datos básicos y el comportamiento de sus operadores aritméticos, de manera que los programas son iguales en todas las plataformas.

- **Multi-hilo.** Java soporta sincronización de múltiples hilos de ejecución (*multithreading*), especialmente útiles en la creación de aplicaciones distribuidas. Así, por ejemplo, mientras un hilo se encarga de la comunicación, otro puede interactuar con el usuario mientras otro presenta una animación en pantalla y otro realiza cálculos.
- **Distribuido.** Java proporciona una colección de clases para su uso en aplicaciones de red, que permiten abrir *sockets* y establecer y aceptar conexiones con servidores o clientes remotos, facilitando así la creación de aplicaciones distribuidas. Además provee interfaces de aplicación tales como RMI e infraestructuras como los *Enterprise Java Beans*, que facilitan el manejo de este tipo de aplicaciones.

Después de analizar las alternativas presentadas, se tomó la decisión de utilizar el lenguaje Java e implementar con él los mecanismos necesarios para realizar la distribución de información en una base de datos distribuida.

Las principales razones por las que se tomó tal decisión se muestran en la figura 6.1.:

| **Utilizar un SMBDD libre** | **Utilizar Java y un SMBD en cada nodo** |
|---|---|
| Una mayor curva de aprendizaje al no haber conocido o utilizado previamente ninguno de los sistemas investigados | Una menor curva de aprendizaje pues ya habíamos manejado el lenguaje y su acceso a bases de datos relacionales |
| El proyecto sólo funcionaría utilizando el sistema de base de datos distribuida adquirido y sería poco escalable, sin la posibilidad de interactuar con bases de datos ya existentes o más conocidas | La existencia del JDBC de Java provee una mayor interacción con diferentes manejadores de bases de datos, libres y propietarios, y la independencia del lenguaje a la plataforma lo hace altamente escalable |
| No se proveería una idea concreta del manejo de transacciones distribuidas, pues éste ya existiría de antemano en el sistema adquirido | El manejo de transacciones distribuidas sería implementado desde cero y se tendría una mejor comprensión de este aspecto esencial de las bases de datos distribuidas |

Figura 6.1. Principales razones por las que se decidió utilizar el lenguaje Java en el desarrollo del proyecto.

Los requerimientos en hardware más importantes con los que se realizaría el proyecto estaban determinados por los tres nodos que componen el *cluster Beowulf* en el que comenzaríamos a trabajar y por la interacción entre ellos, que se muestra en la figura 6.2:

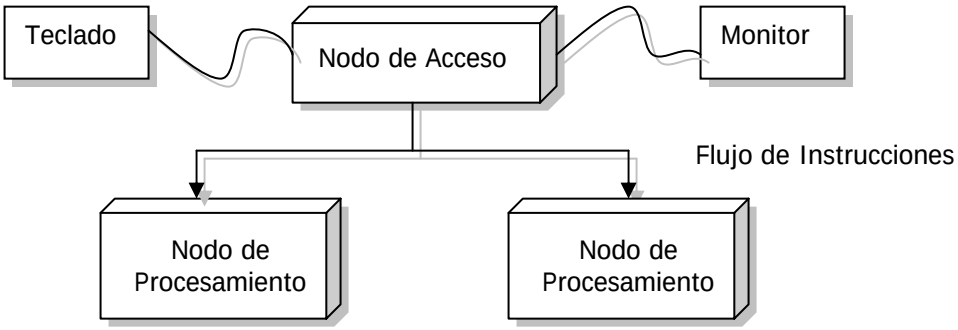



Figura 6.2. Interacción entre los nodos del *Cluster Beowulf*.

Tomando en cuenta que la arquitectura de procesamiento paralelo del *cluster* sigue un paradigma maestro-esclavo, se decidió adoptar la misma estructura para definir la interacción entre los nodos del sistema distribuido, estableciendo como requerimiento fundamental la creación de transacciones distribuidas hacia los tres nodos pero con un manejo de transacciones de tipo centralizado, controlado por el nodo de acceso.

Así, los requerimientos establecidos para el desarrollo del sistema son:

- Utilizar un mecanismo que permita la ejecución remota de procesos en diferentes computadoras.
- Extender el mecanismo para que los procesos remotos puedan ejecutarse en paralelo.
- Utilizar dicho mecanismo para distribuir diferentes sentencias SQL en los tres nodos del *cluster*.
- Crear conexiones a las bases de datos en cada uno de los nodos y ejecutar instrucciones SQL de manera local.
- Integrar en el nodo de acceso el resultado de los procesos realizados sobre las bases de datos en cada uno de los nodos.
- Sincronizar y controlar todos los procesos por medio de una transacción distribuida.
- Obtener los resultados de las sentencias ejecutadas.

## 6. 3. Análisis del Sistema

Con base en el análisis de requerimientos, se estableció el siguiente conjunto de clases candidato que los satisficieran:

- Prueba: Clase que permite probar el funcionamiento del sistema y que representa a una aplicación que hace uso del sistema a desarrollar.
- TransaccionDistribuida: Clase con la que interactúa una aplicación. Recibe de ésta las instrucciones que se deben ejecutar en una transacción y las transforma en un conjunto de colecciones de objetos SQLBean, distribuye dichas colecciones a los nodos de procesamiento y devuelve el resultado del procesamiento total a la clase Prueba.
- ConexionRMI: Interfaz que declara los métodos remotos ofrecidos por un servidor en el sistema de base de datos distribuida.
- ConexionRMI_Impl: Clase que implementa a ConexionRMI. Recibe un conjunto de sentencias a ejecutar en un nodo y las distribuye a las bases de datos correspondientes.
- ConexionBD: Clase que recibe un conjunto de instrucciones y las ejecuta sobre una base de datos.

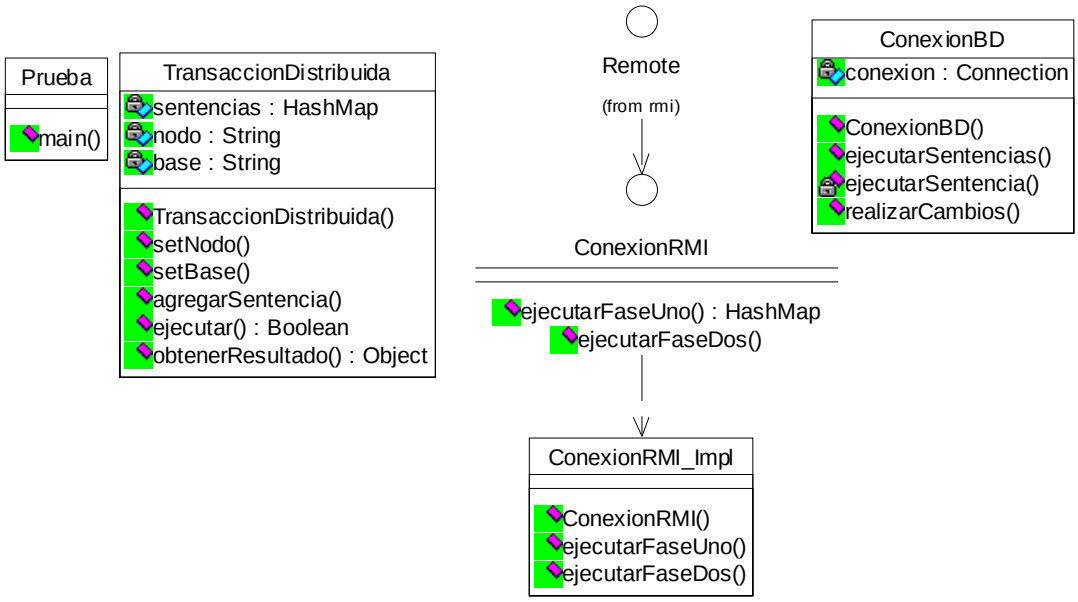



Figura 6.3. Diagrama de clases candidato.

Tomando en cuenta el análisis realizado, en la figura 6.4. se muestra el diagrama de flujo que describe la interacción presentada por las clases candidato para lograr una transacción distribuida:

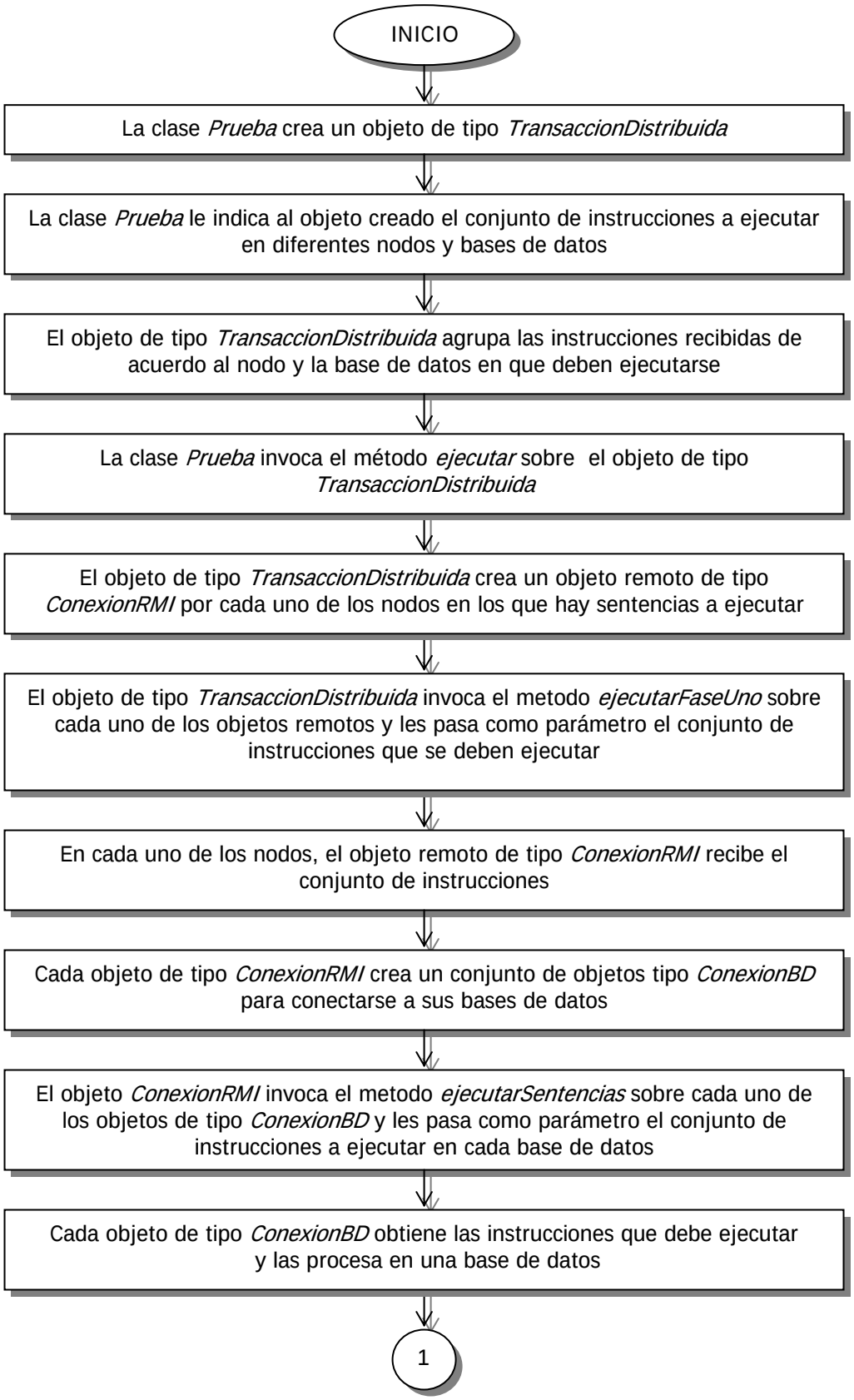



Figura 6.4. Diagrama de flujo de la interacción entre las clases candidato.

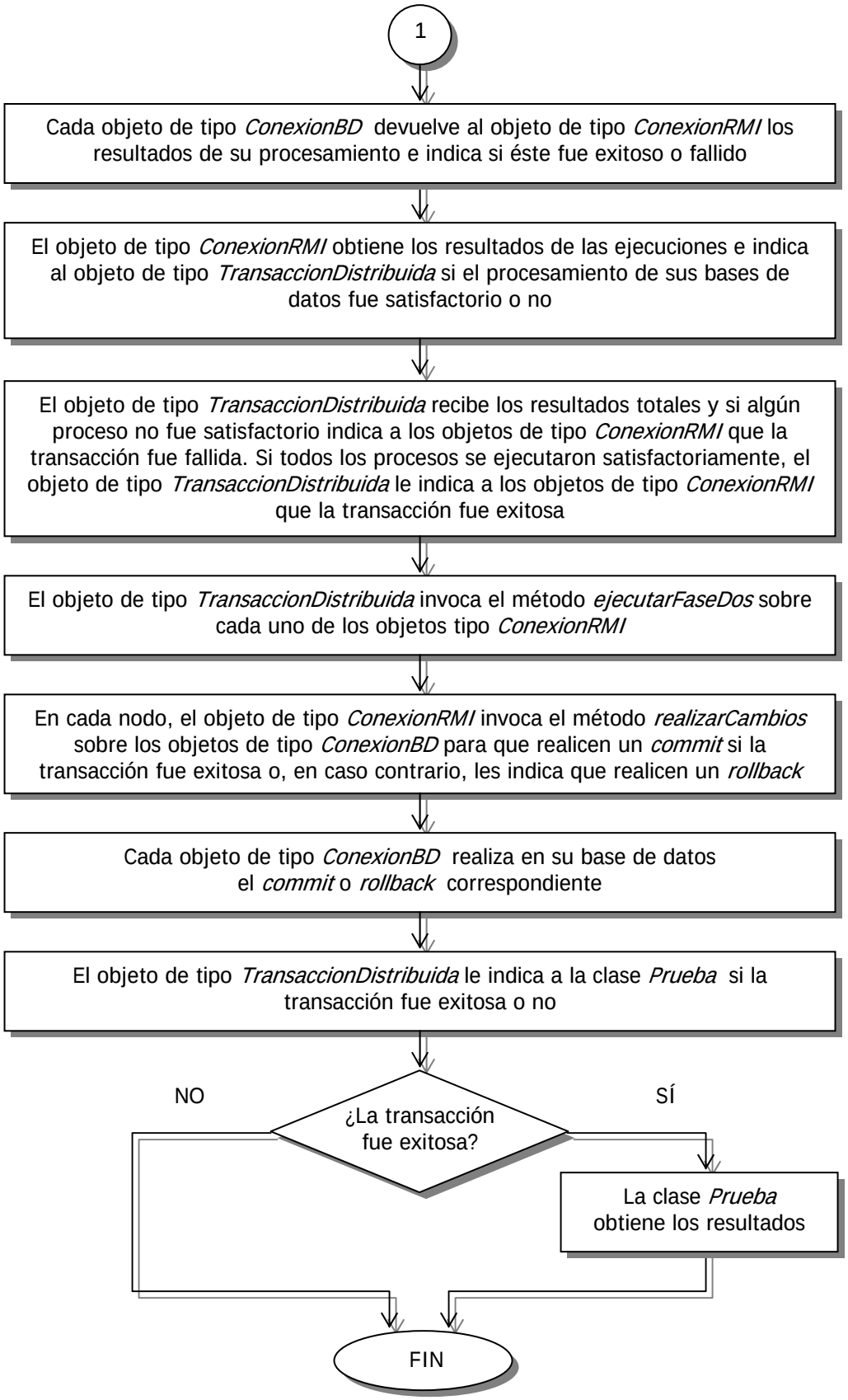



Figura 6.4. (Continuación) Diagrama de flujo de la interacción entre las clases candidato.

De acuerdo con el diagrama de flujo anterior se obtiene el diagrama de secuencia mostrado en la figura 6.5.:

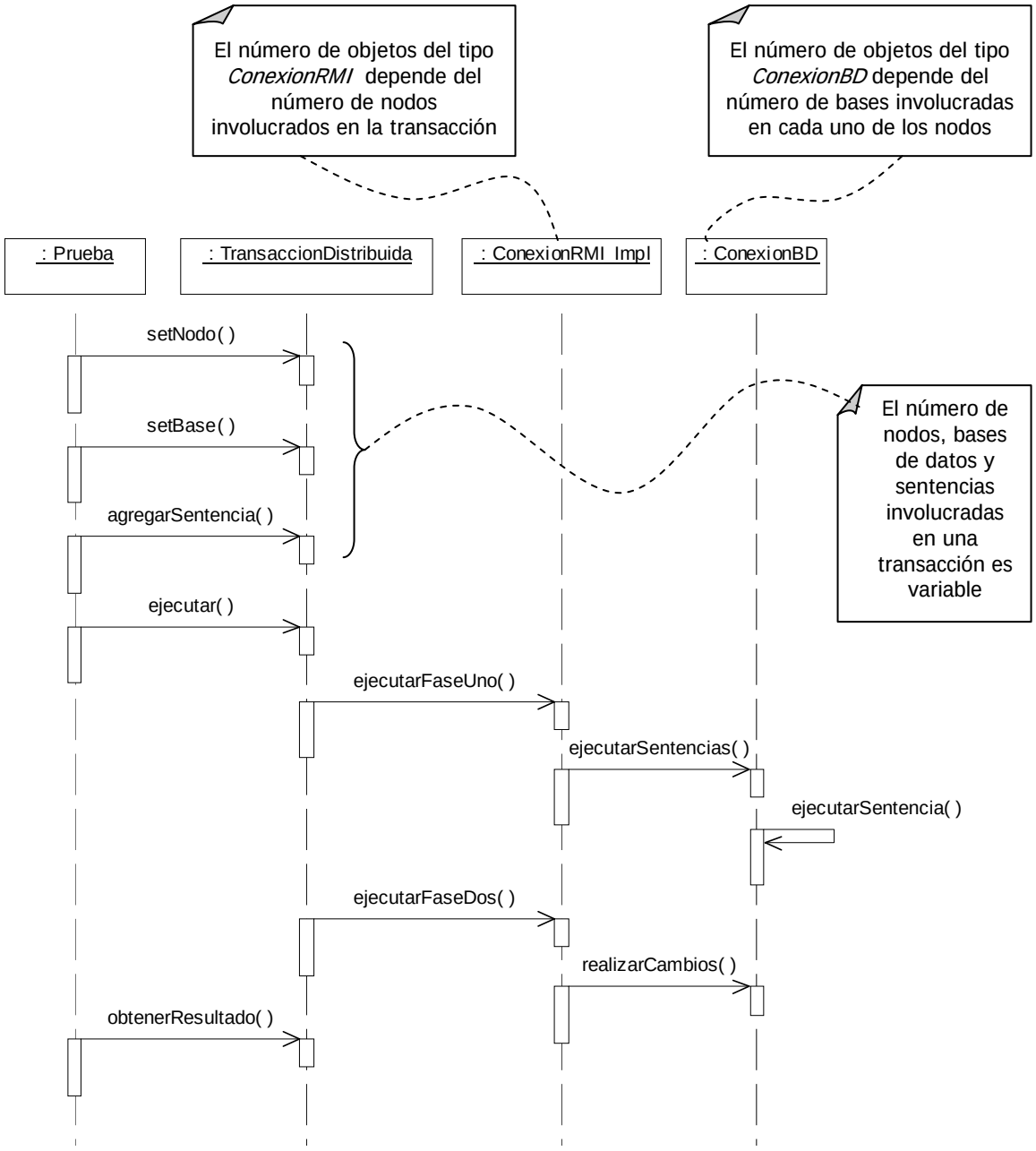



Figura 6.5. Diagrama de flujo de la interacción entre las clases candidato.

De esta manera, se ha modelado en un primer análisis el conjunto de clases que forman el sistema de base de datos distribuida.

### 6. 4. Diseño

De acuerdo con los requerimientos establecidos en la fase de análisis y después de revisar en diversas iteraciones las clases candidato, fue necesario diseñar clases nuevas con propósitos específicos para la transferencia de información a través de la red, la creación de hilos remotos y la creación de hilos para el acceso a bases de datos, así como clases especializadas en la conexión con dichas bases y en la ejecución de sentencias SQL.

Además, el aumento en el número de clases diseñadas implicó la necesidad de agruparlas en distintos paquetes de acuerdo con su localización en los clientes o servidores del sistema y de acuerdo al servicio que proveían, sin embargo, cabe resaltar que las nuevas clases y las modificaciones sufridas por las clases candidato originales no alteraron la esencia de la interacción descrita en el análisis inicial, por lo que los diagramas de flujo y de secuencia explicados anteriormente siguen siendo válidos ya que reflejan el comportamiento general del sistema a desarrollar.

En la figura 6.6 se listan las clases definitivas con las que se desarrolló el sistema de base de datos distribuida y se muestra la manera en que están organizadas:

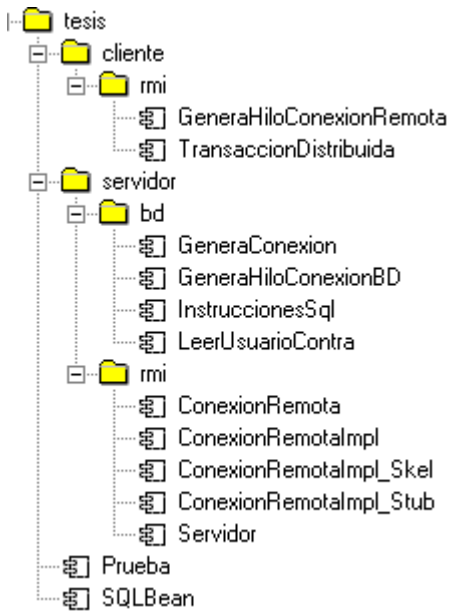



Figura 6.6. Organización de las clases que componen el sistema de base de datos distribuida.

El paquete principal, en el que se encuentran las clases que conforman el manejador de transacciones distribuidas, es el paquete tesis, mostrado en la figura 6.7.

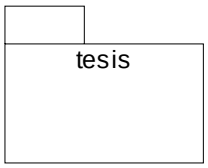



Figura 6.7. Paquete prinicpal del sistema de base de datos distribuida.

El paquete tesis contiene todos los paquetes y clases que forman el manejador de transacciones, así como una clase llamada Prueba que permite probar el funcionamiento del sistema y una clase llamada SQLBean cuyas instancias sirven como objetos de transferencia de datos a través de la red. Esto se muestra en la figura 6.8.

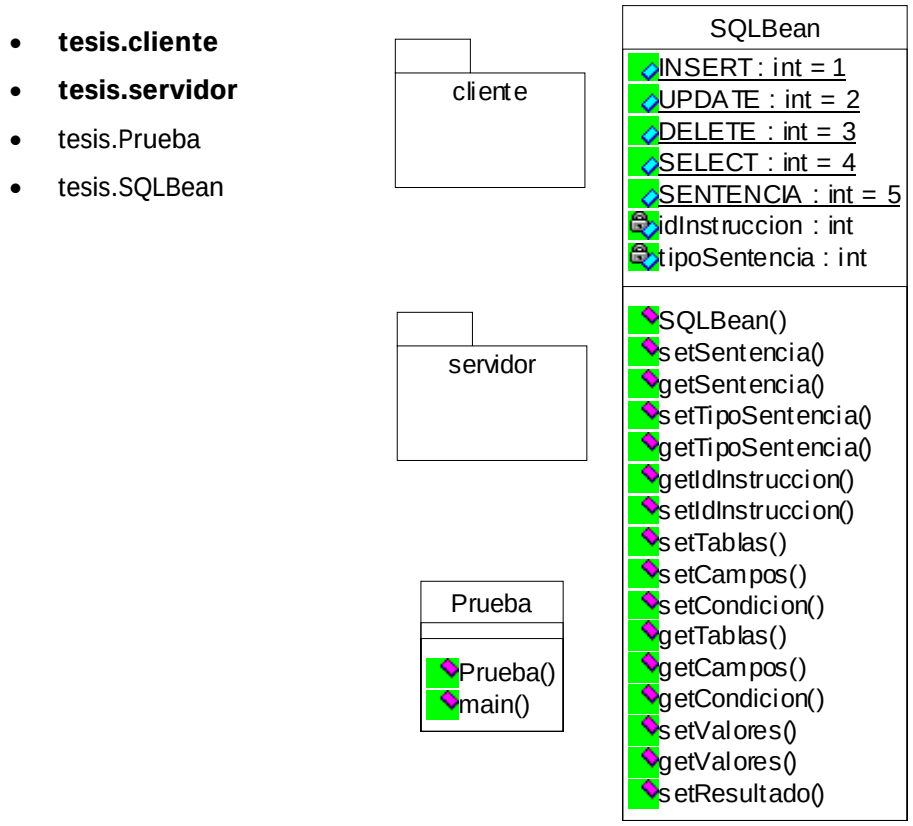



Figura 6.8. Contenido del paquete *tesis*.

Dentro del paquete **tesis.cliente** encontramos el paquete **tesis.cliente.rmi**, mostrado en la figura 6.9, que contiene a las clases necesarias para que una aplicación pueda crear una transacción distribuida y solicitar servicios de bases de datos a otros nodos:

- **tesis.cliente.rmi**


Figura 6.9. Contenido del paquete *tesis.cliente*.

Como lo mostrado en la figura 6.10., dentro del paquete **tesis.cliente.rmi** encontramos siguientes clases:

- TransaccionDistribuida
- GeneraHiloConexionRemota

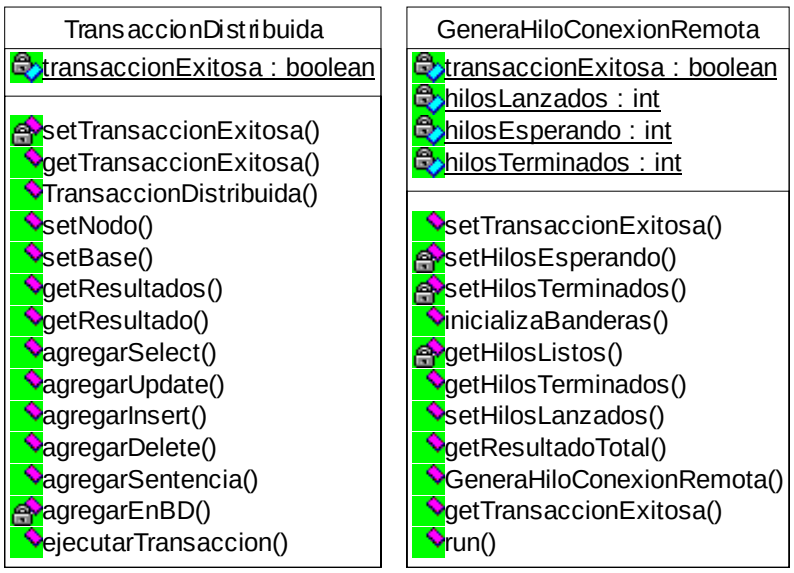



Figura 6.10. Contenido del paquete *tesis.cliente.rmi*.

El paquete **tesis.servidor** contiene los paquetes bd y rmi, como se aprecia en la figura 6.11.

- **rmi**
- **bd**

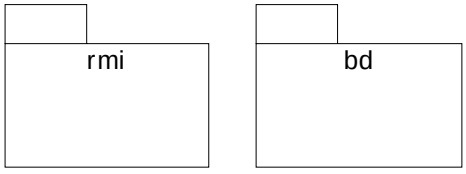



Figura 6.11. Contenido del paquete *tesis.servidor*.

La figura 6.12. muestra el contenido del paquete **tesis.servidor.rmi**:

- Servidor
- ConexionRemota
- ConexionRemotaImpl

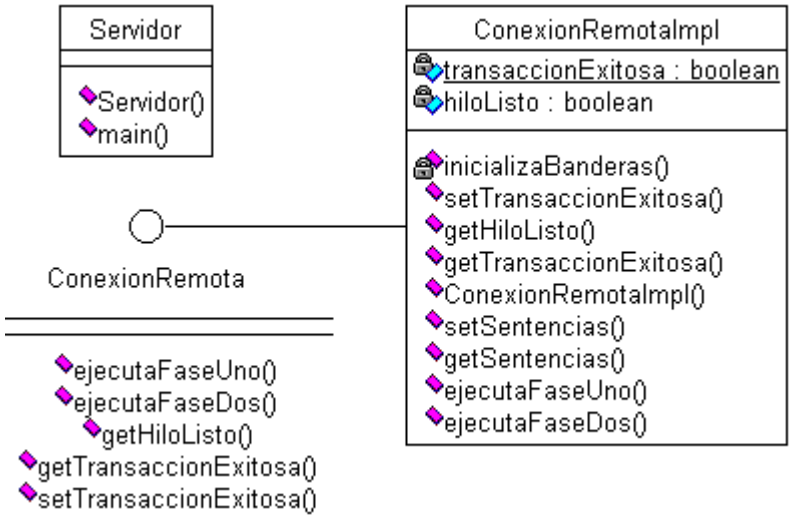



Figura 6.12. Contenido del paquete *tesis.servidor.rmi*.

Finalmente, la figura 6.13. muestra las clases contenidas en el paquete **tesis.servidor.bd**:

- GeneraConexion
- InstruccionesSql
- LeerUsuarioContra

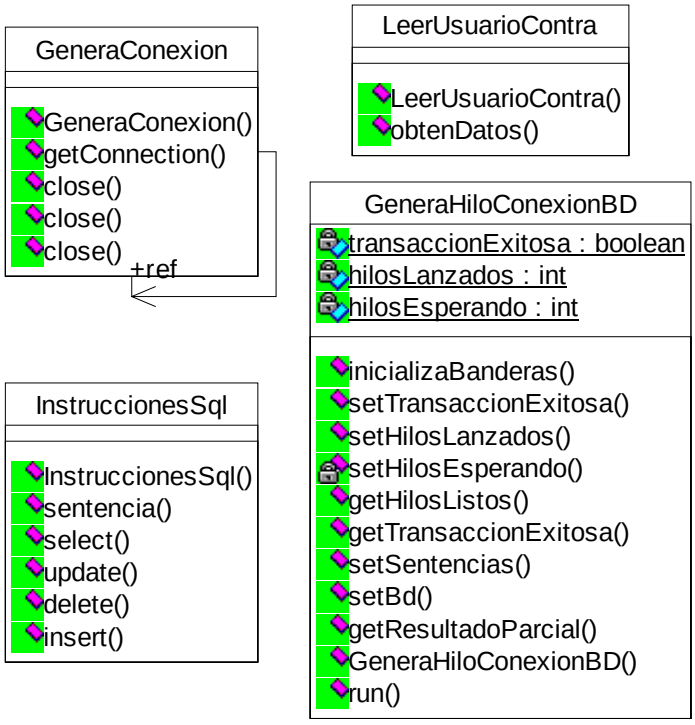



Figura 6.13. Contenido del paquete *tesis.servidor.bd*.

A continuación se listan los paquetes de los que está compuesto este proyecto de tesis y se describe el propósito de las clases de los componen. Cabe resaltar que algunas clases podrían situarse en el nodo cliente para ser utilizadas por alguna aplicación y otras instaladas en los servidores para brindar un servicio de base de datos.

- tesis

  Contiene aquellos paquetes y clases que conforman el presente proyecto, así como una clase que puede utilizarse en el cliente para probar el funcionamiento del sistema.
  - Prueba: Clase que permite probar el funcionamiento del sistema.
  - SQLBean: Clase que transfiere información acerca de una instrucción SQL a ejecutar entre el cliente y los servidores de base de datos.

- tesis.cliente

  Contiene aquellos paquetes que deben encontrarse en el cliente que desee explotar la base de datos distribuida.

- tesis.cliente.rmi

  Contiene aquellas clases que permiten a un cliente solicitar un servicio remoto de base de datos.
  - TransaccionDistribuida: Clase utilizada por alguna aplicación en el cliente para crear una transacción distribuida.
  - GeneraHiloConexionRemota: Clase utilizada para generar tantos hilos de conexión a servidores de bases de datos como requiera la transacción distribuida.
  - ConexionRemotaImpl_Stub: Clase que funge como *stub* para el cliente y permite el envío de mensajes con uno o más servidores. Se genera con el compilador *rmic* a partir de la clase ConexionRemotaImpl.

- tesis.servidor

  Contiene aquellos paquetes que deben encontrarse en cada uno de los nodos servidores que procesan una base de datos dentro del sistema.

- tesis.servidor.rmi

  Contiene aquellas clases que permiten a un nodo brindar un servicio remoto de base de datos para el cliente.

- Servidor: Clase utilizada para registrar un servicio de base de datos y hacerlo disponible a un cliente.
- ConexionRemota: Interfaz utilizada para brindar al cliente el servicio de base de datos que provee un nodo. También debe usarse por el cliente del sistema de base de datos distribuida para obtener un servicio de los servidores a partir del protocolo RMI.
- ConexionRemotaImpl: Implementación de la interfaz ConexionRemota.
- ConexionRemotaImpl_Skel: Clase que funge como *skeleton* para el servidor correspondiente y permite el envío de mensajes con un determinado cliente. Se genera con el compilador *rmic* a partir de la clase ConexionRemotaImpl.

- tesis.servidor.bd

  Contiene aquellas clases que, de manera local, permiten a cada nodo servidor conectarse a la base de datos correspondiente a través de su manejador.
  - GeneraHiloConexionBD: Clase utilizada para generar tantos hilos de conexión a bases de datos como bases de datos requeridas para el servicio que brinda el nodo.
  - GeneraConexion: Clase utilizada para generar una conexión a una base de datos.
  - InstruccionesSQL: Clase que explota una base de datos por medio de la ejecución de instrucciones SQL.
  - LeerUsuarioContra: Clase utilizada para leer el nombre de usuario y la contraseña con la que se accede a las bases de datos en un nodo servidor.

### 6. 5. Implementación

El presente proyecto de tesis comprende el desarrollo de un Sistema de Base de Datos Distribuida (SBDD) implementado en uno de los *cluster*s tipo *Beowulf* con que cuenta la Facultad de Ingeniería. Una de las características principales de esta clase de arquitectura de procesamiento distribuido es el empleo de software libre en todos sus niveles, por lo que se ha adoptado el mismo criterio para la elección del software a utilizar.

De manera general, el proyecto consiste en instalar una base de datos con su manejador en cada uno de los nodos del *cluster* y emplear un lenguaje de programación que permita integrar dichas bases para formar un SBDD. La idea fundamental es programar un Sistema Manejador de Base de Datos Distribuida (SMBDD) que facilite la distribución y manejo de los datos.

### 6. 5. 1. Software utilizado

Una ventaja esencial de utilizar software libre es que no tiene costo alguno, es cada vez más utilizado, está ampliamente difundido y bien documentado. El software que se empleará en el desarrollo del proyecto es:

- MySQL. Es un software de bases de datos con un servidor SQL, programas clientes y bibliotecas, herramientas administrativas, e interfaces de programación (APIs). Es fácil de instalar y administrar. Dispone de una gran cantidad de tutoriales y documentación.
- JAVA. Es un lenguaje de programación de alto nivel orientado a objetos. Cuenta con un compilador que se encarga de convertir el código fuente de un programa en un código intermedio llamado *bytecode* que es independiente de la plataforma en que se trabaje.
- JDBC. Es un API de Java para acceder a sistemas de bases de datos. Está formado por un conjunto de clases e interfaces que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea.
- JAVA RMI. Es el método de invocación remota de Java (Java *Remote Method Invocation*) y hace posible la creación de aplicaciones distribuidas basadas en la tecnología Java.

### 6. 5. 2. Instalación del software

El sistema operativo de los nodos componentes del *cluster* es, en todos los casos, Linux Red Hat, por lo que los procesos de instalación mostrados a continuación sólo son válidos para las distribuciones de dicho sistema. Si se desea instalar el mismo software en computadoras con un sistema operativo distinto o una distribución distinta de Linux, es necesario referirse a la documentación de instalación del software para el caso correspondiente.

Los procesos de instalación descritos en el presente apartado fueron realizados en cada uno de los nodos del *cluster* en el que se desarrolló el presente proyecto de tesis.

#### Instalación de MySQL

Los pasos seguidos para realizar la instalación de MySQL fueron los siguientes:

- Obtención del archivo mysql-standard-4.0.18-pc-linux-i686.tar.gz.
- Instalación de la base de datos:

- Descompresión del archivo mysql-standard-4.0.18-pc-linux-i686.tar.gz:
  tar –xvf mysql-standard-4.0.18-pc-linux-i686.tar.gz
- Movimiento del archivo descomprimido a la carpeta /usr/local:
  mv mysql-standard-4.0.18-pc-linux-i686 /usr/local
- Cambio de localización dentro del árbol de directorios a /usr/local:
  cd /usr/local
- Creación de un enlace simbólico llamado /usr/local/mysql a /usr/local/mysql-standard-4.0.18-pc-linux-i686:
  ln -s mysql-standard-4.0.18-pc-linux-i686 mysql
- Cambio de localización dentro del árbol de directorios a /usr/local/mysql:
  cd mysql
- Creación de un nuevo grupo llamado 'mysql':
  groupadd mysql
- Creación de un nuevo usuario llamado 'mysql' integrante del grupo 'mysql' recientemente creado:
  useradd -g mysql mysql
- Colocación de una copia del archivo my-medium.cnf, ubicado en /usr/local/mysql/support-files, en la carpeta /etc, con el nombre my.cnf:
  cp support-files/my-medium.cnf /etc/my.cnf
- Creación de la base de datos MySQL y los permisos por defecto:
  scripts/mysql_install_db
- Establecimiento del usuario 'root' como propietario de los archivos y directorios ubicados en /usr/local/mysql
  chown -R root .
- Establecimiento del usuario 'mysql' como propietario de los archivos y directorios ubicados en /usr/local/mysql/data
  chown -R mysql data
- Establecimiento del grupo 'mysql' como propietario de los archivos y directorios ubicados en /usr/local/mysql
  chgrp -R mysql .

- Puesta en marcha

  A continuación se muestra la manera en la que se inicia el demonio servidor de MySQL para acceder al manejador de base de datos.

  bin/mysqld_safe &

- Limpieza de la base de datos
    - Cambio de localización dentro del árbol de directorios a /usr/local/mysql/bin:
      cd bin
    - Creación de una clave de acceso (<clave>) para la cuenta MySQL del superusuario 'root':
      ./mysqladmin -u root –p password <clave>
    - Ingreso a la interfaz de línea de comandos de MySQL por medio de la cuenta de superusuario:
      ./mysql -u root -p<clave>
      donde <clave> es la clave de acceso creada en el punto anterior.
      A partir de este momento debe mostrarse el prompt de MySQL: 'mysql>'.
- Eliminación de las cuentas que no poseen claves de acceso.
    - Selección de la base de datos de MySQL:
      mysql> use mysql;
    - Visualización de las cuentas que no poseen una clave de acceso.
      mysql> SELECT user, host, password FROM mysql.user;
    - Eliminación de dichas cuentas.
      mysql> DELETE user FROM mysql.user WHERE password="";
    - Egreso de la interfaz de línea de comandos de MySQL:
      mysql> quit;
    - Término del demonio de MySQL.
      ./mysqladmin -u root -p<clave> shutdown
- Otros comandos
    - Visualización de opciones de ayuda:
      ./mysqld --help
    - Visualización de la versión de MySQL.
      /usr/local/mysql/bin]# ./mysqladmin –V

Para lograr que el demonio servidor de mysql comience su ejecución al momento en que arranca el sistema operativo, realizamos lo siguiente:

- Ubicarnos dentro del directorio en el que está instalado el software de MySQL:
  cd/usr/local/mysql
- Copiar el archivo mysql.server, que se encuentra en el subdirectorio support-files, al directorio /etc/rc.d/init.d
  cp support-files/mysql.server /etc/rc.d/init.d

- Una vez copiado el archivo, necesitamos hacerlo ejecutable, lo que se logra con el siguiente comando:
  chmod +x /etc/rc.d/init.d/mysql.server
- Finalmente, es necesario crear un enlace simbólico al script mysql.server, para que sea ejecutado en tiempo de arranque del sistema operativo. Para esto existen dos posibles ubicaciones, que dependen del modo de arranque del sistema.
  - Si el sistema inicia en modo de consola, debe crearse un enlace simbólico en el siguiente directorio:
    ln -s /etc/rc.d/init.d/mysql.server /etc/rc.d/rc3.d/S98mysql
  - Si el sistema inicia con el modo gráfico de X windows, es necesario teclear lo siguiente:
    ln -s /etc/rc.d/init.d/mysql.server /etc/rc.d/rc5.d/S98mysql

A partir de este momento, el demonio de MySQL comenzará a ejecutarse desde que arranca el sistema operativo y no será necesario iniciarlo de manera manual.

### Instalación del J2SDK

Los pasos seguidos para realizar la instalación del J2SDK fueron los siguientes:

- Obtención del archivo: j2sdk-1_4_0_03-linux-i686-rpm.bin
- Movimiento del archivo j2sdk-1_4_0_03-linux-i686-rpm.bin al directorio de instalación
  mv ./j2sdk-1_4_0_03-linux-i686-rpm.bin /usr/local
- Cambio de ubicación en el sistema de archivos al directorio /usr/local
  cd /usr/local
- Cambio de permisos para convertir el archivo en ejecutable
  chmod a+x j2sdk-1_4_0_03-linux-i686-rpm.bin
- Ejecución
  ./j2sdk-1_4_0_03-linux-i686-rpm.bin
  En este paso se crea el archivo j2sdk-1_4_0_03-linux-i686-rpm
- Instalación del archivo RPM:
  rpm –ivh j2sdk-1_4_0_03-linux-i686-rpm

El J2SDK se instaló en el directorio /usr/local/j2sdk1.4.0_03.

Después de la instalación es necesario configurar algunas variables de ambiente que indiquen al sistema operativo la ubicación de los archivos del J2SDK. Para lograr tal objetivo se agregaron las siguientes líneas de código al final del archivo 'profile' ubicado en la carpeta /etc:

```
PATH=/usr/local/j2sdk1.4.0_03/jre/bin:/usr/local/j2sdk1.4.0_03/jre/lib/i386/client:/usr/local/j2sdk1.
4.0_03/bin:$PATH
export JAVA_HOME=/usr/local/j2sdk1.4.0_03
CLASSPATH=/usr/local/j2sdk1.4.0_03/lib/tools.jar:/usr/local/j2sdk1.4.0_03/jre/lib/rt.jar:/usr/local/j
2sdk1.4.0_03/jre/lib/i386/client:./
```

**Instalación del conector JDBC de Java para la base de datos MySQL**

- Obtención del archivo mysql-connector-java-3.0.11-stable.zip.
- Movimiento del archivo mysql-connector-java-3.0.11-stable.zip al directorio /usr/local
  mv ./mysql-connector-java-3.0.11-stable.zip /usr/local
- Cambio de ubicación en el sistema de archivos al directorio /usr/local
  cd /usr/local
- Descompresión del archivo mysql-connector-java-3.0.11-stable.zip.
  unzip mysql-connector-java-3.0.11-stable.zip

Al descomprimir el archivo se crea el directorio mysql-connector-java-3.0.11-stable y dentro de él se encuentra el archivo mysql-connector-java-3.0.11-stable-bin.jar, que contiene todas las clases que implementan el API JDBC para MySQL. Es necesario que este archivo esté incluido en la variable de ambiente CLASSPATH, declarada dentro del archivo /etc/profile:

```
CLASSPATH=/usr/local/j2sdk1.4.0_03/lib/tools.jar:/usr/local/j2sdk1.4.0_03/jre/lib/rt.jar:/usr/local/j
2sdk1.4.0_03/jre/lib/i386/client:/usr/local/mysql-connector-java-3.0.11-stable/mysql-connector-
java-3.0.11-stable-bin.jar:./
```

### 6. 5. 3. Desarrollo del Sistema

La parte esencial del presente proyecto consiste en desarrollar un Manejador de Transacciones Distribuidas (MTD) que permita ejecutar diferentes sentencias SQL en diversas bases de datos ubicadas en distintos nodos de una red, todo dentro de una misma transacción. Dicho manejador estará formado por una serie de clases e interfaces, desarrolladas con el lenguaje Java

de Sun Microsystems, ubicadas en los nodos cliente y servidores del sistema de base de datos distribuida.

El conjunto de manejadores de base de datos que se encuentran en cada uno de los nodos servidores y el MTD desarrollado conforman el sistema manejador de base de datos distribuida (SMBDD) del presente proyecto. A su vez, el conjunto de las bases de datos contenidas en todos los nodos conforman la Base de Datos Distribuida (BDD). De manera análoga, el Sistema de Base de Datos Distribuida (SBDD) es la unión del SMBDD (Manejador de Transacciones Distribuidas y Manejadores de Base de Datos Locales) junto con la Base de Datos Distribuida o BDD (el conjunto de todas las bases de datos locales).

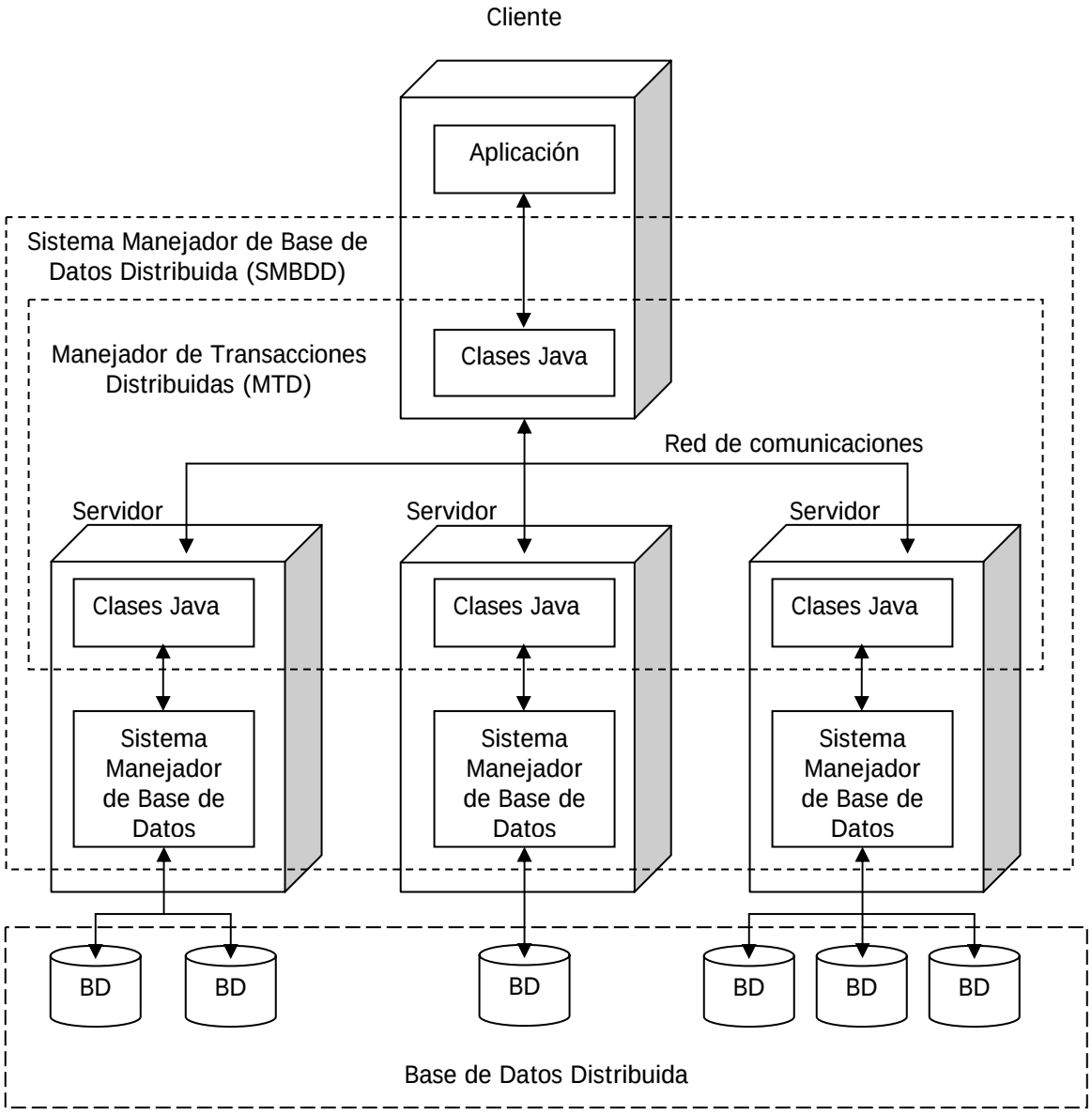



Figura 6.14. Arquitectura del Sistema de Base de Datos Distribuida y sus elementos principales

A continuación se revisará de manera general el código de las clases que componen el manejador de transacciones distribuidas. Se describe la manera en que interactúan dichas clases para lograr una transacción distribuida entre todos los servidores y bases de datos involucradas*.

La interacción entre una determinada aplicación y el sistema de base de datos distribuida se logra a través de la clase TransaccionDistribuida, cuyo constructor recibe la dirección IP y el nombre de la base de datos en la que se ejecutará un conjunto de instrucciones SQL.

Una vez construido un objeto de tipo TransaccionDistribuida, se debe establecer el conjunto de instrucciones a ejecutar en el nodo y la base de datos indicadas en el constructor. Esto se logra a través de cinco diferentes métodos que reciben, como primer argumento, una variable de tipo int que fungirá como identificador global de la instrucción dentro de la transacción a realizar y debe ser único para cada instrucción.

```
public class TransaccionDistribuida
{
……

    public void agregarInsert(int idInsert, String tabla, String campos, String valores){
    ……  }

        public void agregarUpdate(int idUpdate, String tabla, String campos, String valores, String
        condiciones){
    ……  }

    public void agregarDelete(int idDelete, String tabla, String condiciones){
    ……  }

    public void agregarSelect(int idSelect, String tabla, String campos, String condiciones){
    ……          }

    public void agregarSentencia(int idSentencia, String sentencia){
    ……  }
}
```

Los primeros cuatro métodos reciben parámetros específicos para el tipo de instrucción que se desea agregar a la transacción (insert, update, delete o select). El método agregarSentencia es una generalización, y recibe como segundo argumento una cadena que representa una

* Se utilizarán los caracteres …… para indicar partes donde existe código irrelevante para la explicación, tal como métodos get y set de propiedades, manejo de excepciones, etc.

sentencia SQL a ejecutar sin importar su tipo y es utilizada cuando se requiere de una estructura más compleja que en las instrucciones SQL simples.

Una vez agregadas las sentencias SQL a ejecutar en el nodo y la base de datos indicadas en el constructor, es posible agregar nuevas instrucciones a nodos y bases de datos diferentes.

```
public class TransaccionDistribuida
{
…….

        public void setNodo(String nodo){
…….          }

        public void setBase(String base){
…….          }

  }
```

El método setNodo permite indicar una nueva dirección IP correspondiente al nodo que procesará las instrucciones SQL que se agreguen posteriormente. Para indicar la base de datos específica en la que se ejecutarán dichas instrucciones se utiliza el método setBase. De esta manera se puede agregar cualquier número de sentencias SQL a ejecutar en diferentes nodos y bases de datos dentro de una misma transacción.

A continuación se muestra un código de ejemplo, en el que una aplicación crea un objeto de tipo TransaccionDistribuida y establece un conjunto de instrucciones a ejecutar en diferentes nodos y bases de datos:

```
public class AplicacionDeEjemplo{

    public static void main(String args[]){

        TransaccionDistribuida td = new TransaccionDistribuida("170.70.9.52","baseAdmin");
        td.agregarInsert(1,"USUARIOS", " nombre, clave"," 'Jesus','xxx' ");
        td.agregarUpdate(2,"USUARIOS", "clave='nuevaClave' ", "nombre = 'Claudia' ");
        td.setNodo("170.70.9.50");
        td.setBase("baseUsuarios");
        td.agregarDelete(3, "DATOS_USUARIO", " nombre = 'Desconocido' '");
        td.agregarSelect(4,"DATOS_USUARIO", " * ", " nombre = 'Laura' ");
        td.setNodo("170.70.9.51");
        td.setBase("baseCatalogos");
        td.agregarSentencia(5," select id, valor from CAT_DEPARTAMENTOS where id > 100 ");
        …….
        if (td.ejecutarTransaccion() ){
```

```
        // Obtener resultados
         }
      }
   }
```

De acuerdo con lo comentado, en la figura 6.15 se muestra la interacción entre una determinada aplicación y el sistema de base de datos distribuida:

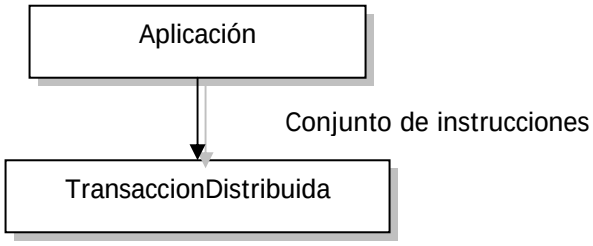



Figura 6.15. Una aplicación establece el conjunto de instrucciones que debe ejecutar la clase TransaccionDistribuida.

De manera interna, la clase TransaccionDistribuida organiza las sentencias SQL a ejecutar en una estructura jerárquica, separándolas de acuerdo con los nodos y las bases de datos que les corresponden.

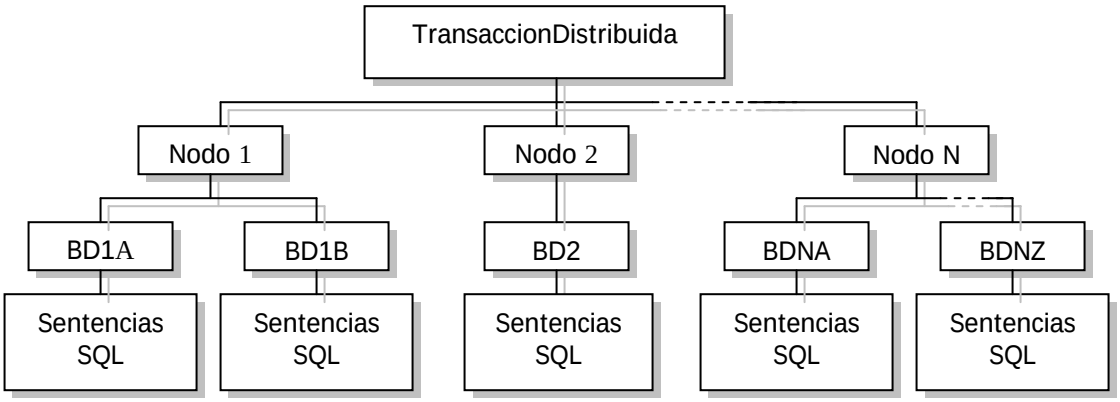



Figura 6.16. Organización jerárquica de sentencias SQL a ejecutar dentro de la clase TransaccionDistribuida

Una vez que todas las instrucciones han sido indicadas se debe invocar el método ejecutarTransaccion(). En ese momento, la clase TransaccionDistribuida crea un hilo distribuido de conexión a cada uno de los servidores a los que se les solicitará un servicio de base de datos, de acuerdo con lo establecido por la aplicación que la utiliza. Cada hilo se crea utilizando la clase

GeneraHiloConexionRemota() cuyo constructor recibe dos parámetros: la *ip* del nodo para el que se creará un objeto remoto y el conjunto de instrucciones que se procesarán en las bases de datos contenidas en dicho nodo.

```
public class TransaccionDistribuida{
……..

    public  boolean ejecutarTransaccion(){
        ……..
        Iterator it = nodos.iterator();
        ……..
        while (it.hasNext()){
            ……..
            GeneraHiloConexionRemota hcr= new GeneraHiloConexionRemota(ip,bases);
            Thread hiloRemoto = new Thread(hcr);
            hiloRemoto.start();
        }
    }
}
```

Al invocar el método start() sobre cada uno de los hilos creados, se ejecuta el método run() de la clase GeneraHiloConexionRemota, en donde se crea propiamente un objeto remoto de tipo ConexionRemota y se invoca el método remoto ejecutaFaseUno(), que corresponde conceptualmente a la fase uno del c*ommit* de dos fases, indicando el conjunto de instrucciones que se deben ejecutar sobre las bases de datos de dicho nodo por cada uno de los diferentes hilos.

```
public class GeneraHiloConexionRemota implements Runnable{
……..
    public  void run(){
        ……..
        // Se crea un objeto remoto de conexión a un servidor y se indican las instrucciones
        // a ejecutar en sus bases de datos
        ConexionRemota c = (ConexionRemota)
       Naming.lookup("rmi://"+ip+":1099/ServicioConexionRemota");
        c.ejecutaFaseUno(bases);
        }
}
```

En la figura 6.17 se puede apreciar de manera gráfica la manera en que se crean los hilos distribuidos:

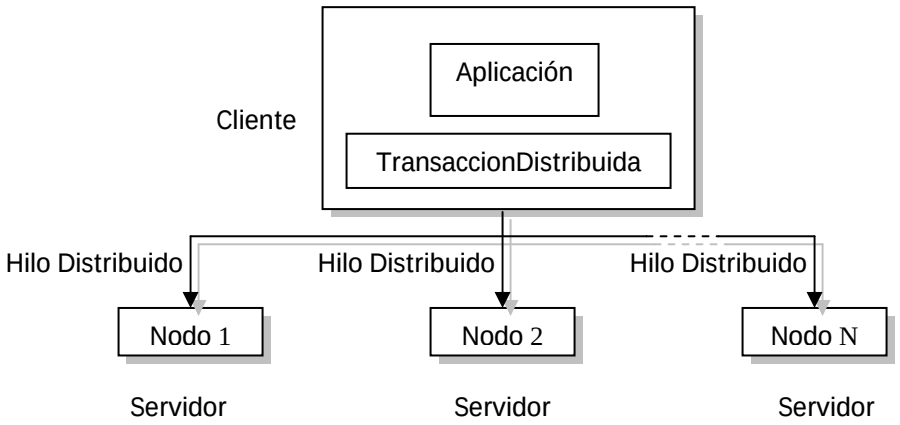



Figura 6.17. Creación de hilos y objetos remotos
por la clase TransaccionDistribuida

El método ejecutaFaseUno( ) de la interfaz ConexionRemota es ejecutado por la clase ConexionRemotaImpl, que la implementa en el servidor de base de datos correspondiente. En dicho método, se crea un hilo de conexión a cada una de las bases de datos locales involucradas en la transacción distribuida. Esto se logra utilizando un objeto de tipo GeneraHiloConexionBD, cuyo constructor recibe el nombre de la base de datos a la que se debe conectar y el conjunto de instrucciones a ejecutar. De manera semejante a los hilos creados por la clase TransaccionDistribuida, al invocar el método start() se ejecuta el método run() de la clase GeneraHiloConexionBD, donde se ejecutan las instrucciones en las bases de datos correspondientes.

```
public class GeneraHiloConexionBD implements Runnable {
……
      public static final int INSERT =1;
      public static final int UPDATE =2;
      public static final int DELETE =3;
      public static final int SELECT =4;
      public static final int SENTENCIA =5;


    public GeneraHiloConexionBD(String bd, Collection sentencias){
        this.bd = bd;
        this.sentencias = sentencias;
        ……
    }
```

```
public void run(){
        ……..
        // Se crea una conexión a la base de datos correspondiente
        Connection conn= GeneraConexion.getConnection(bd);
        InstruccionesSql isql= new InstruccionesSql(conn);
        Iterator itSql = sentencias.iterator();
        while (itSql.hasNext()) {
        ……..
        // Evaluamos qué tipo de sentencia se va a ejecutar
        switch (tipoSentencia) {
                case INSERT:{
                    ……..
                    resultado = isql.insert(t,campos,valores);
                    break; }
                case UPDATE:{
                    ……..
                    resultado = isql.update(t,campos, condiciones);
                    break; }
                case DELETE:{
                    ……..
                    resultado = isql.delete(t,condiciones);
                    break; }
                case SELECT:{
                    ……..
                    resultado = isql.select(t,campos,condiciones);
                    break; }
                case SENTENCIA:{
                    ……..
                    resultado=isql.sentencia(sentencia);
                    break; }
                default: {
         break;}
           }
……..
       }
       ……..
   }
……..
}
```

El método ejecutaFaseUno( ) es auxiliado por dos clases principales: GeneraConexion e InstruccionesSQL. La primera se encarga de generar una conexión a la base de datos indicada y la segunda es utilizada para ejecutar las diferentes instrucciones sql de la transacción que corresponden a la base de datos sobre la que se estableció la conexión.

```
public class GeneraConexion{
……..
    public static GeneraConexion ref=new GeneraConexion();

    // Constructor donde se obtiene el driver de acceso al manejador de la BD, de acuerdo con JDBC
    public GeneraConexion(){
        ……..
        Class.forName("com.mysql.jdbc.Driver");
          ……..
     }

    public static Connection getConnection(String bd) throws SQLException {
        // Leemos de un archivo de configuración el usuario y la contraseña para generar la conexión
        ……..
        LeeUsuarioContra lda =new LeeUsuarioContra();
        datos=lda.obtenDatos();
        String url = "jdbc:mysql://localhost/"+bd;
        String usu=datos[0];
        String pass=datos[1];
        Connection conn = DriverManager.getConnection(url,usu,pass);
        // Establecemos una propiedad de la base de datos que permita realizar commits
        // y rollbacks en tablas transaccionales
        ……..
        if (conn.getAutoCommit()){
            conn.setAutoCommit(false);
        }
    return conn;
    }
}
```

La clase LeerUsuarioContra es una clase muy simple que se encarga de leer un archivo de configuración y obtener el nombre de usuario y contraseña con que se accede a cada una de las bases de datos.

La clase InstruccionesSQL cuenta con diversos métodos para ejecutar diferentes instrucciones SQL. En su método constructor recibe el objeto de conexión a la BD correspondiente. Los diferentes métodos con los que cuenta hacen uso de dicha conexión y reciben los parámetros necesarios para ejecutar la instrucción adecuada.

```
public class InstruccionesSql{

    private Connection con;

    public InstruccionesSql(Connection con){
        this.con = con;
    }
```

```
public synchronized Object sentencia(String sentencia) throws SQLException{
    boolean resultset = false;
    Object resultado = null;
    ……
    Statement st = con.createStatement();
    resultset = st.execute(sentencia);
    if (!resultset){
        resultado = new Integer(st.getUpdateCount());
    } else {
        ResultSet resp = st.getResultSet();
        ArrayList datos = new ArrayList();
        ResultSetMetaData rsmd=resp.getMetaData();
        int numCols=rsmd.getColumnCount();
        while (resp.next()) {
            Vector renglon = new Vector();
            for (int i=1; i<=numCols;i++){
                renglon.addElement(resp.getString(i));
            }
            datos.add(renglon);
        }
        resp.close();
        st.close();
        resultado = datos;
    }
return resultado;
}

public ArrayList select(String tablas,String campos,String condiciones) throws SQLException {
    ……
    if(condiciones==null || condiciones.equals("")){
        sql = "Select "+campos+" from "+tablas;
    else
        sql = "Select "+campos+" from "+tablas+" where "+condiciones;
    resultado = (ArrayList) sentencia(sql);
    ……
    return resultado;
}


public Integer update(String tablas,String campos,String condiciones) throws SQLException{
    ……
    if(condiciones==null  || condiciones.equals(""))
        sql="Update "+tablas+" set "+campos;
    else
        sql="Update "+tablas+" set "+campos+" where "+condiciones;
    resultado = (Integer) sentencia(sql);
    ……
    return resultado;
}
```

```
    public Integer delete(String tablas,String condiciones) throws SQLException  {
        ……
        if(condiciones==null  || condiciones.equals(""))
            sql="DELETE FROM"+tablas;
        else
            sql="DELETE FROM "+tablas+" WHERE "+condiciones;
        resultado = (Integer) sentencia(sql);
        ……
        return resultado;
    }

    public Integer insert(String tabla,String campos,String values) throws SQLException  {
        ……
        if(condiciones==null  || condiciones.equals(""))
            sql="Insert into "+tabla+" values ("+values+")";
        else
            sql="insert into "+tabla+" ("+campos+")  values("+values+")";
        resultado = (Integer) sentencia(sql);
        ……
        return resultado;
    }
}
```

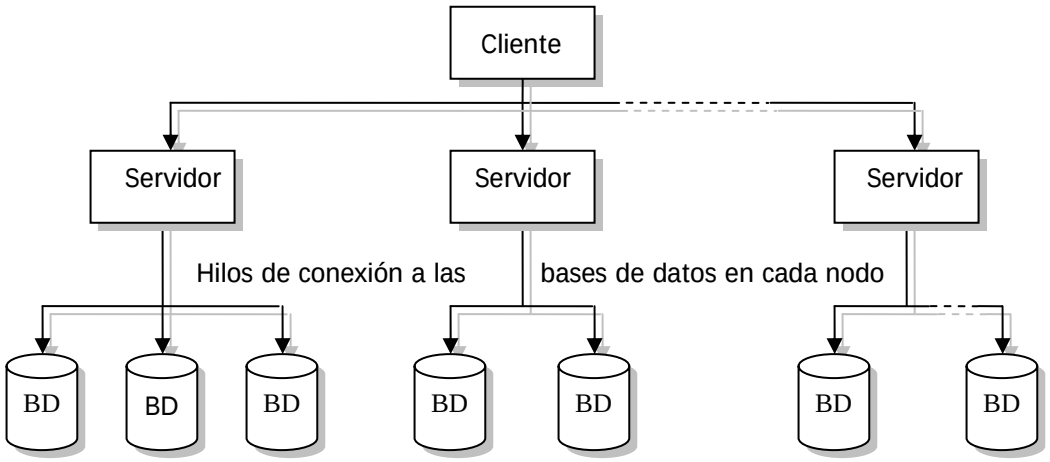



Figura 6.18. Creación de hilos locales de conexión a las bases de datos por cada uno de los servidores y ejecución de las sentencias SQL

Hasta este punto, se han ejecutado todas las instrucciones SQL en las bases de datos y los nodos involucrados en una transacción distribuida. De acuerdo con el protocolo c*ommit* de dos Fases, es necesario que cada uno de los nodos indique al manejador de transacciones (propiamente la clase TransaccionDistribuida) si el resultado de su ejecución fue exitoso o no.

Para explicar esto revisemos parte del código restante en la clase GeneraHiloConexionBD:

```
public class GeneraHiloConexionBD implements Runnable {
……..

    public void run(){
        try {
            ……..
            // Aquí se encuentra el código explicado anteriormente,  donde se ejecutan
            // todas las instrucciones asignadas a una cierta base de datos
            ……..
        } catch (SQLException e){
            // Si ocurre alguna falla, se indica que la ejecución de las operaciones en el nodo correspondiente
            // no fue exitosa, si no ocurre falla alguna la transacción local en las bases del nodo es exitosa
            ……..
            setTransaccionExitosa(false);
        } finally {
            // Todos los hilos de conexión a base de datos suspenden su ejecución en espera
            // de que el manejador de transacciones les indique el estado global de la transacción
            synchronized(Thread.currentThread()){
            ……..
            Thread.currentThread().wait();
            }
            ……..
        }
        ……..
    }
}
```

Las instrucciones asignadas en cada nodo han sido ejecutadas y ya se tiene una respuesta de falla o éxito por cada uno de ellos. La clase ConexionRemotaImpl, que implementa a la interfaz ConexionRemota, es la encargada de indicar al manejador de transacciones el resultado total de las ejecuciones en cada uno de los nodos.

```
public class ConexionRemotaImpl extends java.rmi.server.UnicastRemoteObject implements
                                                                    ConexionRemota {
……..

    public void ejecutaFaseUno(HashMap parametros) throws java.rmi.RemoteException{
    ……..
        // En este punto es donde se crearon los hilos de conexión a las diferentes bases de datos
        // que posee el nodo correspondiente y se ejecutaron las instrucciones SQL asignadas
        ……..
        //El proceso actual espera a que todos los hilos de conexión a base de datos hayan ejecutado
        // las sentencias correspondientes para saber el estado total de la transacción en el nodo servidor
        while (!GeneraHiloConexionBD.getHilosListos()){ }
        ……..
        // Se considera entonces un valor booleano que indica si hubo fallas o no en el servidor,
        // para informar a la clase GeneraHiloConexionRemota que se encuentra en el cliente.
```

```
        this.setTransaccionExitosa(GeneraHiloConexionBD.getTransaccionExitosa());
        if (this.getTransaccionExitosa()){
            // Si hubo éxito almacena los resultados en una estructura de datos
            resultado = GeneraHiloConexionBD.getResultadoParcial();
        }
        ……
    }
        // Informa al cliente que se han realizado todas las operaciones asignadas
        // en las bases de datos del nodo servidor donde se ejecuta el proceso
        hiloListo=true;
    }
……
}
```

Por su parte, el nodo cliente espera a que la ejecución de todas las operaciones en los diferentes nodos servidores hayan terminado y verifica si hubo algún error o si todos los procesos fueron ejecutados exitosamente.

```
public class GeneraHiloConexionRemota implements Runnable{
……

    public  void run(){
        ……
        try {
            c = (ConexionRemota) Naming.lookup("rmi://"+ip+":1099/ServicioConexionRemota");
            // En esta parte se creó un objeto remoto de conexión a un servidor de base de datos
            // y se invocó el método ejecutarFaseUno
            ……
        } catch (Exception murle) {
                // Si hubo alguna falla, la transacción se indica como fallida
                setTransaccionExitosa(false);
        } finally {
        ……


          // Esperamos a que el nodo servidor haya terminado de procesar todas sus bases de datos
          while (!c.getHiloListo()){    }
         // Una vez que el servidor ha terminado verificamos si el proceso que realizó
         //  fue exitoso o no y almacenamos su estado en la variable que indica el estado
         // global de la transacción
        setTransaccionExitosa(c.getTransaccionExitosa());
        ……
        // Esperamos a que todos los servidores hayan terminado su ejecución
        while (!getServidoresListos()){    }
         // Una vez que todos los servidores han terminado su ejecución les informamos
        // si la transacción global fue exitosa o no, dependiendo de si algún nodo reportó una falla
        c.setTransaccionExitosa(getTransaccionExitosa());
        if (this.getTransaccionExitosa()){
            // Si la transacción fue exitosa almacenamos los resultados enviados
```

```
                // por cada uno de los servidores
                HashMap resultado = GeneraHiloConexionBD.getResultadoParcial();
                if (resultado!=null){
                   synchronized(this.getClass()){
                     resultadoTotal.putAll(resultado);
                   }
                }
                ……..
            }
            // Invocamos el método ejecutarFaseDos para que todos los nodos realicen un commit o
            // un rollback en sus respectivas bases, de acuerdo con el estado global de la transacción
            c.ejecutaFaseDos();
            ……..
        }
        // Informamos a la clase TransaccionDistribuida que se ha terminado de procesar la transacción
        setHilosTerminados();
    }
}
```

Todos los nodos han informado al manejador de transacciones el estado de su procesamiento local de base de datos. Si existió alguna falla en algún nodo, el nodo cliente informa al resto de los nodos que la transacción no fue exitosa, de acuerdo con la segunda fase del c*ommit* de dos fases.

El método ejecutarFaseDos en cada uno de los servidores despierta a los hilos locales de conexión a las bases de datos para que realicen un *commit* o un *rollback*, de acuerdo con lo indicado por el manejador de transacciones.

```
public class ConexionRemotaImpl extends java.rmi.server.UnicastRemoteObject implements
ConexionRemota {
……..

    public HashMap ejecutaFaseDos() throws java.rmi.RemoteException {
        ……..
        // Todos los hilos de conexión a base de datos que se encuentran suspendidos
        // son recuperados y reactivados
        Thread al[] = new Thread[numHilos];
        tg.enumerate(al,false);
        for (int k=0; k<al.length; k++){
        Thread t = al[k];
        synchronized(t){
           // El método notify despierta a cada uno de los hilos de conexión a las bases de datos locales
           t.notify();
        }
      }
      ……..
  }
```

```
}
```

Por su parte, el hilo de conexión cuenta ya con el resultado de la transacción global y, dependiendo de dicho estado, realiza el *commit* o *rollback* correspondiente.

```
public class GeneraHiloConexionBD implements Runnable {
……..

    public void run(){
       try {
           // En esta parte se ejecutaron las sentencias SQL de la fase uno
       } catch (SQLException e){
         // Si hubo alguna falla se indicó que la transacción local en el nodo no fue exitosa
         setTransaccionExitosa(false);
       } finally {
           synchronized(Thread.currentThread()){
                // El hilo fue suspendido en espera de la indicación por el manejador de transacciones
                Thread.currentThread().wait();
           }

           // El hilo despierta en la segunda fase y evalúa si la transacción fue exitosa o fallida
           // para realizar un commit o un rollback en cada base de datos local
           if (transaccionExitosa){
               isql.sentencia("COMMIT");
           } else {
              isql.sentencia("ROLLBACK");
           }
       }
    }
}
```

De esta manera, implementando el protocolo del c*ommit* de dos fases, se logra construir una transacción distribuida entre diferentes bases de datos comunicadas por una red.

Si la transacción fue exitosa, la aplicación que utiliza la clase TransaccionDistribuida puede obtener los resultados de las sentencias procesadas a través de sus identificadores globales:

```
    public class Prueba{

        public static void main(String args[]){

            TransaccionDistribuida td = new TransaccionDistribuida("170.70.9.52","baseAdmin");
            td.agregarInsert(1,"USUARIOS", " nombre, clave"," 'Jesus','xxx' ");
            td.agregarUpdate(2,"USUARIOS", "clave" ," 'nuevaClave' ", "nombre = 'Claudia' ");
            td.setNodo("170.70.9.50");
            td.setBase("baseUsuarios");
            td.agregarDelete(3, "DATOS_USUARIO", " nombre = 'Desconocido' '");
            td.agregarSelect(4,"DATOS_USUARIO", " * ", " nombre = 'Laura' ");
            td.setNodo("170.70.9.51");
```

```
        td.setBase("baseCatalogos");
        td.agregarSentencia(5," select id, valor from CAT_DEPARTAMENTOS where id > 100 ");
        ……
        if (td.ejecutarTransaccion() ){
       ArrayList resultadoSelect1 = (ArrayList) td.getResultado(4);
       ArrayList resultadoSelect2 = (ArrayList) td.getResultado(5);
       Integer numeroDeRegistrosEliminados = (Integer) td.getResultado(3);
        }
    }
}
```

### 6. 5. 4. Compilación de las clases desarrolladas

- Nos ubicamos en el directorio donde se encuentra el código fuente.
  cd /home/tesisSBDD/
- Utilizamos el compilador de java y compilamos todas las clases
  javac –d . *.java
  La opción -d . indica al compilador que además de compilar las clases cree los paquetes necesarios.
- Utilizamos el compilador de RMI para crear el *stub* y el *skeleton* de la clase ConexionRemotaImpl, que se encuentra en el paquete tesis.servidor.rmi
  *rmic* tesis.servidor.rmi.ConexionRemotaImpl

### 6. 5. 5. Puesta en marcha del SBDD

- En cada uno de los nodos servidores abrimos una consola y damos de alta el servicio de registro RMI, en el directorio donde se encuentran las clases compiladas
  rmiregistry &
- Ejecutamos el proceso servidor que permitirá al cliente explotar las bases de datos contenidas en el nodo correspondiente
  java tesis.servidor.rmi.Servidor

  Aparecerá el siguiente mensaje:
  Iniciando Servidor de Base de Datos
  Servidor de Base de Datos listo para procesar solicitudes

- En el nodo cliente ejecutamos la aplicación que haga uso del sistema de base de datos distribuida, en nuestro caso, la clase Prueba.
  java tesis.AplicacionDeEjemplo

En ese momento, se creará el objeto de tipo TransaccionDistribuida con instrucciones SQL a ejecutar en diferentes servidores y bases de datos. Se crearán los hilos distribuidos por cada servidor y, de manera paralela, cada uno procesará las instrucciones asignadas en sus bases de datos, de acuerdo con el protocolo *commit* de dos fases. Finalmente, si la transacción es exitosa y si los resultados son mostrados en consola por la aplicación, podrán verse los datos obtenidos.

De manera alternativa, podrá revisarse el estado de cada base de datos involucrada para verificar la correcta modificación de los datos, ya sea que se haya realizado un *commit* o un *rollback* en todas y cada una de las bases, de acuerdo con el estado global de la transacción.

# RESULTADOS Y ANÁLISIS DEL DESEMPEÑO DEL SISTEMA DE BASE DE DATOS DISTRIBUIDA

El presente proyecto tiene como finalidad comprobar el procesamiento en una base de datos distribuida construida en un *cluster* tipo *Beowulf*, cumpliendo con las características esenciales de una base de datos distribuida, usando software libre y tomando en cuenta los recursos con los que se cuenta.

Debido a lo anterior se hicieron pruebas del procesamiento distribuido de base de datos en los tres nodos que componen el *cluster* de acuerdo con lo siguiente:

- **Pruebas de procesamiento paralelo en comparación con el nodo de acceso.**

La primera serie de pruebas realizadas consistió en evaluar el tiempo de procesamiento de base de datos variando el número de unidades de procesamiento y de bases de datos accedidas.

Para evaluar el desempeño del sistema en un solo nodo se utilizó el nodo de acceso al *cluster* que, como se describió anteriormente, posee una mayor capacidad de procesamiento que los dos nodos restantes. Éstos fueron utilizados en las pruebas que involucraban dos procesadores (el nodo de acceso y un nodo de procesamiento) y finalmente los tres nodos procesando diferentes bases de datos. En cada uno de los casos se midió el desempeño del procesamiento utilizando primero una sola base de datos por nodo y después dos bases de datos por cada uno.

Para la ejecución de diferentes transacciones con sentencias INSERT distribuidas entre los diferentes nodos y bases de datos se obtuvieron los siguientes resultados:

| Transacciones con sentencias INSERT | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **3000** | **3000** | **8.3** |
| **1** | **2** | **1500** | **1500** | **8.7** |
| 2 | 1 | 1500 | 1500 | 20.9 |
| 2 | 2 | 750 | 750 | 21.8 |
| 3 | 1 | 1000 | 1000 | 14.8 |
| 3 | 2 | 500 | 500 | 15.3 |

Para la ejecución de diferentes transacciones con sentencias SELECT distribuidas entre los diferentes nodos y bases de datos se obtuvieron los siguientes resultados:

| Transacciones con sentencias SELECT | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **1** | **3000** | **4.3** |
| **1** | **2** | **1** | **1500** | **4.7** |
| 2 | 1 | 1 | 1500 | 11.6 |
| 2 | 2 | 1 | 750 | 12.1 |
| 3 | 1 | 1 | 1000 | 8.5 |
| 3 | 2 | 1 | 500 | 8.9 |

Para la ejecución de diferentes transacciones con sentencias UPDATE distribuidas entre los diferentes nodos y bases de datos se obtuvieron los siguientes resultados:

| Transacciones con sentencias UPDATE | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **3000** | **3000** | **7.5** |
| **1** | **2** | **1500** | **1500** | **7.6** |
| 2 | 1 | 1500 | 1500 | 18.4 |
| 2 | 2 | 750 | 750 | 19.0 |
| 3 | 1 | 1000 | 1000 | 15.1 |
| 3 | 2 | 500 | 500 | 14.4 |

Para la ejecución de diferentes transacciones con sentencias DELETE distribuidas entre los diferentes nodos y bases de datos se obtuvieron los siguientes resultados:

| Transacciones con sentencias DELETE | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **3000** | **3000** | **6.4** |
| **1** | **2** | **1500** | **1500** | **6.6** |
| 2 | 1 | 1500 | 1500 | 12.3 |
| 2 | 2 | 750 | 750 | 12.6 |
| 3 | 1 | 1000 | 1000 | 10.2 |
| 3 | 2 | 500 | 500 | 10.6 |

De los resultados anteriores se puede observar que al realizar el procesamiento paralelo del mismo número de registros afectados en total para 1, 2 y 3 nodos de procesamiento no se obtuvo una disminución en la ejecución de las transacciones distribuidas comparadas con las transacciones efectuadas en el nodo de acceso del *cluster*.

Contrariamente a lo esperado, el tiempo de ejecución de las transacciones era mayor cuando los datos se encontraban distribuidos que cuando eran procesados por un único nodo.

Se verificó que el nodo de acceso al *cluster* cuenta con una mayor velocidad de procesamiento (procesador Pentium 4 a 2.4 GHz) que no era superada por el procesamiento realizado en los tres nodos de manera paralela. El resto de los nodos cuenta con un procesador Pentium Pro a 180 MHz. Dado que estos dos nodos cuentan con procesadores menos potentes se llegó a la conclusión de que producían un retraso en el tiempo de procesamiento distribuido comparado contra el tiempo de procesamiento del primer nodo.

- **Pruebas de procesamiento paralelo en comparación con un nodo diferente al de acceso.**

Para verificar si el tiempo de procesamiento en paralelo era reducido en comparación con el tiempo de procesamiento de uno de los nodos más lentos, se midió el tiempo de respuesta en otro de los nodos del *cluster*, para las mismas transacciones realizadas en el nodo de acceso, obteniendo los siguientes resultados:

| Transacciones con sentencias INSERT | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **3000** | **3000** | **24.7** |
| **1** | **2** | **1500** | **1500** | **25.5** |
| 2 | 1 | 1500 | 1500 | 20.9 |
| 2 | 2 | 750 | 750 | 21.8 |
| 3 | 1 | 1000 | 1000 | 14.8 |
| 3 | 2 | 500 | 500 | 15.3 |

| Transacciones con sentencias SELECT | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **1** | **3000** | **14.2** |
| **1** | **2** | **1** | **1500** | **14.5** |
| 2 | 1 | 1 | 1500 | 11.6 |
| 2 | 2 | 1 | 750 | 12.1 |
| 3 | 1 | 1 | 1000 | 8.5 |
| 3 | 2 | 1 | 500 | 8.9 |

| Transacciones con sentencias UPDATE | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **3000** | **3000** | **22.5** |
| **1** | **2** | **1500** | **1500** | **23.8** |
| 2 | 1 | 1500 | 1500 | 18.4 |
| 2 | 2 | 750 | 750 | 19.0 |
| 3 | 1 | 1000 | 1000 | 15.1 |
| 3 | 2 | 500 | 500 | 14.4 |

| Transacciones con sentencias DELETE | | | | |
|---|---|---|---|---|
| Nodos | BD por nodo | Sentencias por BD | Registros afectados por BD | Tiempo promedio [s] |
| **1** | **1** | **3000** | **3000** | **20.2** |
| **1** | **2** | **1500** | **1500** | **21.6** |
| 2 | 1 | 1500 | 1500 | 12.3 |
| 2 | 2 | 750 | 750 | 12.6 |
| 3 | 1 | 1000 | 1000 | 10.2 |
| 3 | 2 | 500 | 500 | 10.6 |

Tomando estos datos en cuenta y verificando los tiempos de respuesta cuando los datos están distribuidos y el procesamiento se realiza de manera paralela utilizando dos y tres nodos, se puede apreciar que sí existe una reducción en el tiempo de procesamiento ya que, de los nodos restantes, uno tiene las mismas características y el otro es más potente que cualquiera de ellos, por lo que el procesamiento en paralelo no implica ningún retraso.

# CONCLUSIONES

El presente proyecto de tesis ha involucrado el desarrollo de un manejador de transacciones distribuidas escrito en lenguaje Java que, junto con los manejadores de base de datos locales en cada uno de los servidores y sus bases de datos, conforma un sistema de base de datos distribuida.

A continuación se mencionan los logros obtenidos en el desarrollo del sistema de base de datos distribuida comprendido en el presente proyecto de tesis:

El manejador de transacciones distribuidas desarrollado coordina la ejecución de sentencias SQL en diferentes servidores y bases de datos.

La coordinación de todas las sentencias a ejecutar se realiza mediante el protocolo *commit de dos fases.*

La utilización del protocolo *commit de dos fases* hace posible la creación de una transacción distribuida entre las bases de datos comunicadas por una red en el *cluster Beowulf* de la Facultad de Ingeniería, pudiéndose extender a una red de computadoras de mayor amplitud.

La utilización del lenguaje Java hace que el sistema de base de datos distribuida desarrollado sea independiente del sistema operativo con que cuenta cualquiera de los nodos involucrados, tanto el cliente como los servidores, aunque en nuestro caso todos contaban con Linux RedHat.

Por simplicidad se ha utilizado MySQL en todos los servidores del presente proyecto aunque podría no ser el caso, pues la utilización del API JDBC de Java permite que el sistema de base de datos distribuida desarrollado sea independiente de las bases de datos utilizadas.

El API RMI es soportado por la versión estándar del lenguaje Java, eliminando la necesidad de utilizar versiones más complejas, como la versión empresarial de dicho lenguaje.

En contraste con la utilización de los Enterprise Java Beans de la versión empresarial de Java, la utilización de Java RMI elimina la necesidad de contar con un servidor de aplicaciones en

cada uno de los nodos procesadores de las bases de datos y permite una programación más sencilla.

El sistema de base de datos desarrollado no se encuentra ligado a algún modelo de base de datos específico, por lo que puede ser utilizado para explotar cualquier conjunto de bases de datos relacionales, independientemente de la estructura y los datos que contengan.

El sistema de base de datos distribuida puede ser utilizado por cualquier aplicación escrita en lenguaje Java, como una aplicación *standalone*, una aplicación gráfica o una aplicación web.

La interacción de una aplicación con el sistema de base de datos distribuida se realiza a través de la clase TransaccionDistribuida. Con sólo nueve métodos (cinco para agregar sentencias, uno para cambiar de nodo, uno para cambiar de base, uno para ejecutar y uno para obtener resultados) se puede indicar, de un modo sencillo, la manera en la que se desea crear una transacción distribuida y obtener sus resultados.

Dado que el método ejecutarTransaccion de la clase TransaccionDistribuida devuelve un booleano que indica si la transacción es exitosa o no, un programador que utilice el sistema de base de datos distribuida puede tomar las decisiones necesarias en la lógica de su aplicación en caso de algún error sin que ésta se vea afectada gravemente.

El sistema de base de datos distribuida desarrollado es fácilmente escalable ya que el número de servidores y bases de datos involucradas en una transacción distribuida puede variar de acuerdo con los recursos disponibles.

Mediante el manejo de excepciones de java, así como de excepciones remotas proporcionado por Java RMI y excepciones de base de datos del API JDBC, el sistema de base de datos distribuida presenta tolerancia a fallas de comunicación en la red y de procesamiento de bases de datos en todos y cada uno de los nodos y las bases de datos involucradas.

Las pruebas de desempeño mostraron una mejoría en los tiempos de procesamiento de base de datos para dos de los nodos del *cluster* que cuentan con las mismas características, no así para uno de los tres nodos, cuya velocidad de procesamiento era evidentemente mayor a la presentada por el resto de los nodos con los que colaboraba.

El sistema desarrollado en este proyecto de tesis ha confirmado de manera práctica que un sistema de base de datos distribuida puede presentar un mejor desempeño que el encontrado en un sistema centralizado.

El sistema también permite que una aplicación acceda a diversas fuentes de datos de manera homogénea a través de transacciones globales conseguidas por medio de la implementación del protocolo *commit de dos fases*.

De esta manera, la investigación realizada en el presente proyecto ha brindado las bases teóricas para el desarrollo de un sistema de base de datos distribuida y éste, a su vez, ha permitido verificar de manera práctica el procesamiento de distintas bases de datos distribuidas en una red de computadoras, además de soportar tolerancia a fallas por medio del manejo de excepciones en Java, siendo un sistema altamente portable, genérico y escalable.

Por estas razones, concluimos que se han cumplido de manera satisfactoria los objetivos fijados al inicio de la investigación, tanto a nivel práctico como teórico.

# BIBLIOGRAFÍA Y REFERENCIAS


- Kroenke, David. Database processing: fundamentals, design, implementation. MacMillan Publishing Company. Cuarta Edición. Nueva Jersey, EUA. 1995.
  http://www.emunix.emich.edu/~khailany/645/ch15_Kroenke9.ppt

- Tamer, Özsu y Valduriez, Patrick. Principles of Distributed Database Systems. Prentice Hall. Segunda Edición. 1999.
  http://www.cs.ualberta.ca/~database/ddbook.html

- Balderas Jiménez, Roy; Moreno Sánchez, Jeanette; Plácido Mojica, Martín. Administración del cluster tipo Beowulf de la Facultad de Ingeniería. Tesis Profesional. Facultad de Ingeniería. UNAM. 2003.

- Farley, Jim. Java Distributed Computing. O'Reilly & Associates, Inc. Primera Edición. 1998.
  http://borg.mi.infn.it/libri/books/javaenterprise/dist/index.htm

- Monson-Haefel, Richard. Enterprise JavaBeans. O'Reilly & Associates, Inc. Segunda Edición. 1999.
  http://borg.mi.infn.it/libri/books/javaenterprise/ebeans/index.htm

- Flanagan, David, et al. Java Enterprise in a nutshell: A desktop quick reference. O'Reilly & Associates, Inc. Primera edición. 1999.
  http://borg.mi.infn.it/libri/books/javaenterprise/jenut/index.htm

- Arnold, Ken y Gosling, James. El lenguaje de programación Java. Addison-Wesley, Domo. Primera Edición. 1997.

- Página del Cinvestav con información acerca de los sistemas de base de datos distribuidas.
  http://www.cs.cinvestav.mx/SC/prof_personal/adiaz/Disdb/temario.html

- Sitio oficial de la tecnología Java, de Sun Microsystems.
  http://java.sun.com

- Sitio oficial del sistema de base de datos MySQL.
  http://www.mysql.com

- Sitio con información acerca de la interacción entre Java y la base de datos MySQL, utilizando JDBC.
  http://www.dei.inf.uc3m.es/docencia/p_s_ciclo/avbd/web/apuntes/jdbc.doc#_Toc87324754

- "Metodologías de paralelización en la Supercomputadora CICESE2000" . Sitio con información acerca de arquitecturas de computadoras.
  http://telematica.cicese.mx/computo/super/cicese2000/paralelo/

- Página del Posgrado de Ciencias e Ingeniería en Computación de la UNAM, donde se brinda una descripción general del modelo de objetos distribuidos Java RMI.
  http://www.mcc.unam.mx/~cursos/Algoritmos/javaDC99-2/RMI1.html

- Menchaca, Rolando y García, Félix. Java RMI. Artículo publicado en la Revista Digital Universitaria. Vol. 2 No.1.
  http://www.revista.unam.mx/vol.2/num1/art3/

- Curso teórico y práctico de la tecnología Java RMI.
  http://www.infor.uva.es/~cllamas/sd/rmi/jgurututut/RMI.html

- Página del Ing. Carlos Alberto Román Zamitiz, de la Faculta de Ingeniería de la UNAM, donde se brinda información concerniente a la programación orientada a objetos con Java, así como al análisis y diseño orientados a objetos con el Lenguaje Unificado de Modelado UML.
  http://www.fi-b.unam.mx/pp/profesores/carlos/

- Página con un catálogo de sistemas de bases de datos de software libre.
  http://www.faqs.org/faqs/databases/free-databases

# APÉNDICE

## I. ConexionRemota.java

```
package tesis.servidor.rmi;

import java.sql.*;
import java.io.Serializable;
import java.util.*;
import tesis.servidor.rmi.*;

public interface ConexionRemota extends java.rmi.Remote
{
  public void ejecutaFaseUno(HashMap datos) throws java.rmi.RemoteException, java.sql.SQLException;
  public HashMap ejecutaFaseDos() throws java.rmi.RemoteException, java.sql.SQLException;
  public boolean getHiloListo() throws java.rmi.RemoteException;
  public boolean getTransaccionExitosa() throws java.rmi.RemoteException;
  public void setTransaccionExitosa(boolean transaccionExitosaA) throws java.rmi.RemoteException;

}
```

## II. ConexionRemotaImpl.java

```
package tesis.servidor.rmi;

import java.sql.*;
import java.io.Serializable;
import java.util.*;
import tesis.servidor.bd.*;

public class ConexionRemotaImpl extends java.rmi.server.UnicastRemoteObject implements
ConexionRemota {

        HashMap sentencias;
        HashMap resultado = new HashMap();
        HashMap conexionesBD;
        private static boolean transaccionExitosa=true;
        private boolean hiloListo=false;
        private ThreadGroup tg = null;

        private void inicializaBanderas(){
                transaccionExitosa=true;
                hiloListo=false;
                tg = null;
        }
```

```
    public void setTransaccionExitosa(boolean transaccionExitosaA) throws java.rmi.RemoteException{
      synchronized(this.getClass()){
        if (transaccionExitosa){
          transaccionExitosa = transaccionExitosaA;
          GeneraHiloConexionBD.setTransaccionExitosa(transaccionExitosaA);
        }
      }
    }

    public boolean getHiloListo() throws java.rmi.RemoteException{
      return hiloListo;
    }

    public boolean getTransaccionExitosa() throws java.rmi.RemoteException{
      return transaccionExitosa;
    }

    public ConexionRemotaImpl() throws java.rmi.RemoteException {
   super();
      resultado = new HashMap();
    }

    public void setSentencias(HashMap sentencias){
      this.sentencias = sentencias;
    }

    public HashMap getSentencias(){
      return sentencias;
    }

    public void ejecutaFaseUno(HashMap parametros) throws java.rmi.RemoteException,
java.sql.SQLException{
      if (parametros!=null){
        setSentencias(parametros);
        HashMap resultadoParcial = new HashMap();
        Set nombresBases = sentencias.keySet();
        Iterator it = nombresBases.iterator();
        ArrayList arrGhcbd = new ArrayList();
        tg = new ThreadGroup("TG_SBDD");
        int numHilos =sentencias.size();
        GeneraHiloConexionBD.setHilosLanzados(numHilos);
        while (it.hasNext() && GeneraHiloConexionBD.getTransaccionExitosa()){
          String bd = (String) it.next();
          Collection sentenciasSQL = (ArrayList) sentencias.get(bd);
          GeneraHiloConexionBD ghcbd = new GeneraHiloConexionBD(bd, sentenciasSQL);
          arrGhcbd.add(ghcbd);
          Thread hiloConexionBd = new Thread(tg,ghcbd,"H("+bd+")");
          hiloConexionBd.start();
        }
        while (!GeneraHiloConexionBD.getHilosListos()){
        }
        this.setTransaccionExitosa(GeneraHiloConexionBD.getTransaccionExitosa());
        if (this.getTransaccionExitosa()){
          resultado = GeneraHiloConexionBD.getResultadoParcial();
        } else {
          resultado = null;
```

```
      }
    }
    hiloListo=true;
  }

  public HashMap ejecutaFaseDos() throws java.rmi.RemoteException, java.sql.SQLException{
    try {
      int numHilos = getSentencias().size();
        tg.list();
        Thread al[] = new Thread[numHilos];
        tg.enumerate(al,false);
        for (int k=0; k<al.length; k++){
          Thread t = al[k];
          synchronized(t){
            t.notify();
          }
        }
      } catch (Exception e){
        e.printStackTrace();
      } finally {
        while (tg.activeCount()>0){
        }
        GeneraHiloConexionBD.inicializaBanderas();
        inicializaBanderas();
      }
    return resultado;
    }
  }
```

## III. datos.txt

```
login:root
password:38B-AW56
```

## IV. GeneraConexion.java

```
package tesis.servidor.bd;

import java.sql.*;

public class GeneraConexion
{
  public static GeneraConexion ref=new GeneraConexion();


  public GeneraConexion()   {
    try{
      Class.forName("com.mysql.jdbc.Driver");
    }       catch(ClassNotFoundException e){
      e.printStackTrace();
    }
   }
```

```
  public static Connection getConnection(String bd) throws SQLException {
    String datos[]=new String[2];
    LeerUsuarioContra lda =new LeerUsuarioContra();
    datos=lda.obtenDatos();
    String url = "jdbc:mysql://localhost/"+bd;
    String usu=datos[0];
    String pass=datos[1];
    Connection conn = DriverManager.getConnection(url,usu,pass);
    if (conn.getAutoCommit()){
      conn.setAutoCommit(false);
    }
    return conn;
  }

  public static void close(ResultSet rs){
    try{
      rs.close();
    }catch(Exception e){
      e.printStackTrace();
    }
  }

  public static void close(Statement stm){
    try{
      stm.close();
    }catch(Exception e){
      e.printStackTrace();
    }
  }

  public static void close(Connection conn){
    try{
      conn.close();
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

## V. GeneraConexionBD.java

```
package tesis.servidor.bd;

import java.sql.*;
import java.io.Serializable;
import java.util.*;
import tesis.SQLBean;

public class GeneraHiloConexionBD implements Runnable {

  private Collection sentencias;
  private String bd;
  private static boolean transaccionExitosa=true;
```

```
private static int hilosLanzados=0;
private static int hilosEsperando=0;
private static HashMap resultadoParcial = new HashMap();

public static void inicializaBanderas(){
  transaccionExitosa=true;
  hilosLanzados=0;
  hilosEsperando=0;
}

public static void setTransaccionExitosa(boolean transaccionExitosaA){
  try {
    if (transaccionExitosa){
      transaccionExitosa = transaccionExitosaA;
    }
  } catch (Exception e){
    transaccionExitosa=false;
  }
}

public static void setHilosLanzados(int hilos){
  hilosLanzados = hilos;
}

private void setHilosEsperando(){
  synchronized(this.getClass()){
    ++hilosEsperando;
  }
}

public static boolean getHilosListos(){
  if (hilosLanzados==hilosEsperando && hilosLanzados>0){
    return true;
  } else {
    return false;
  }
}

public static boolean getTransaccionExitosa(){
  return transaccionExitosa;
}


public void setSentencias(Collection sentencias){
  this.sentencias = sentencias;
}

public void setBd(String bd){
  this.bd = bd;
}

public static HashMap getResultadoParcial(){
  return resultadoParcial;
}
```

```
public GeneraHiloConexionBD(String bd, Collection sentencias){
  this.bd = bd;
  this.sentencias = sentencias;
  resultadoParcial = new HashMap();
}

public void run(){
  Connection conn=null;
  InstruccionesSql isql= null;
  try {
    conn=GeneraConexion.getConnection(bd);
    isql= new InstruccionesSql(conn);
    Iterator itSql = sentencias.iterator();
    while (itSql.hasNext()) {
      SQLBean sqlB = (SQLBean) itSql.next();
      int idSentencia = sqlB.getIdInstruccion();
      Object resultado = null;
      int tipoSentencia = sqlB.getTipoSentencia();
      switch (tipoSentencia) {
        case SQLBean.INSERT:{
          String t = sqlB.getTablas();
          String campos = sqlB.getCampos();
          String valores = sqlB.getValores();
          resultado = isql.insert(t,campos,valores);
          break;
        }
        case SQLBean.UPDATE:{
          String t = sqlB.getTablas();
          String campos = sqlB.getCampos();
          String condiciones = sqlB.getCondicion();
          resultado = isql.update(t,campos, condiciones);
          break;
        }
        case SQLBean.DELETE:{
          String t = sqlB.getTablas();
          String condiciones = sqlB.getCondicion();
          resultado = isql.delete(t,condiciones);
          break;
        }
        case SQLBean.SELECT:{
          String t = sqlB.getTablas();
          String campos = sqlB.getCampos();
          String condiciones = sqlB.getCondicion();
          resultado = isql.select(t,campos,condiciones);
          break;
        }
        case SQLBean.SENTENCIA:{
          String sentencia = sqlB.getSentencia();
          resultado=isql.sentencia(sentencia);
          break;
        }
        default: {
          break;
        }
       }
      synchronized(this.getClass()){
```

```
            if (resultado!=null){
              resultadoParcial.put(new Integer(idSentencia), resultado);
            }
          }
        }
      } catch (SQLException e){
        e.printStackTrace();
        setTransaccionExitosa(false);
      } finally {
        try {
          synchronized(Thread.currentThread()){
            setHilosEsperando();
            Thread.currentThread().wait();
          }
          if (transaccionExitosa){
            isql.sentencia("COMMIT");
          } else {
            isql.sentencia("ROLLBACK");
          }
        } catch(Exception e){
          e.printStackTrace();
        }
      }
    }
  }
```

## VI. GeneraHiloConexionRemota.java

```
package tesis.cliente.rmi;

import java.sql.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import java.io.Serializable;
import java.util.*;
import javax.naming.*;
import tesis.servidor.rmi.*;


public class GeneraHiloConexionRemota implements Runnable{

  private static HashMap resultadoTotal = new HashMap();
  private HashMap bases;
  private String ip;
  private static boolean transaccionExitosa=true;
  private static int hilosLanzados=0;
  private static int hilosEsperando=0;
  private static int hilosTerminados=0;
```

```
public void setTransaccionExitosa(boolean transaccionExitosaA){
  synchronized(this.getClass()){
    if (transaccionExitosa){
      transaccionExitosa = transaccionExitosaA;
    }
  }
}

private void setHilosEsperando(){
  synchronized(this.getClass()){
    ++hilosEsperando;
  }
}

private void setHilosTerminados(){
  synchronized(this.getClass()){
    ++hilosTerminados;
  }
}

public static void inicializaBanderas(){
  transaccionExitosa=true;
  hilosLanzados=0;
  hilosEsperando=0;
  hilosTerminados=0;
}


private static boolean getHilosListos(){
  if (hilosLanzados==hilosEsperando && hilosLanzados>0){
    return true;
  } else {
    return false;
  }
}

public static boolean getHilosTerminados(){
  if (hilosLanzados==hilosTerminados && hilosLanzados>0){
    return true;
  } else {
    return false;
  }
}

public static void setHilosLanzados(int hilos){
    hilosLanzados = hilos;
}

public static HashMap getResultadoTotal(){
  return resultadoTotal;
}
```

```
  public GeneraHiloConexionRemota(String ip, HashMap bases){
    this.ip=ip;
    this.bases = bases;
    resultadoTotal = new HashMap();
    transaccionExitosa=true;
  }

  public static boolean getTransaccionExitosa(){
    return transaccionExitosa;
  }

  public void run(){
    ConexionRemota c = null;
    try {
      c = (ConexionRemota) Naming.lookup("rmi://"+ip+":1099/ServicioConexionRemota");
      c.ejecutaFaseUno(bases);
    } catch (Exception murle) {
      murle.printStackTrace();
      setTransaccionExitosa(false);
    } finally {
      try {
        while (!c.getHiloListo()){
        }
        setTransaccionExitosa(c.getTransaccionExitosa());
        setHilosEsperando();
        while (!getHilosListos()){
        }
        c.setTransaccionExitosa(getTransaccionExitosa());
        if (this.getTransaccionExitosa()){
          HashMap resultado = c.ejecutaFaseDos();
          if (resultado!=null){
            if (null!=resultadoTotal){
              synchronized(this.getClass()){
                resultadoTotal.putAll(resultado);
              }
            }
          }
        } else {
          synchronized(this.getClass()){
            resultadoTotal = null;
          }
        }
      } catch (Exception murle) {
        murle.printStackTrace();
      }
    }
    setHilosTerminados();
  }
}
```

## VII. InstruccionesSql.java

```
package tesis.servidor.bd;

import java.sql.*;
import java.io.*;
import java.util.*;

public class InstruccionesSql{

  Connection con;

  public InstruccionesSql(Connection con){
    this.con=con;
  }

  public synchronized Object sentencia(String sentencia) throws SQLException{
    boolean resultset = false;
    Object resultado = null;
    try {
      Statement st = con.createStatement();
      synchronized(this) {
        resultset = st.execute(sentencia);
      }
      if (!resultset){
        resultado = new Integer(st.getUpdateCount());
      } else {
        ResultSet resp = st.getResultSet();
        ArrayList datos = new ArrayList();
        ResultSetMetaData rsmd=resp.getMetaData();
        int numCols=rsmd.getColumnCount();
        while (resp.next()){
          Vector renglon = new Vector();
          for (int i=1; i<=numCols;i++){
            renglon.addElement(resp.getString(i));
          }
          datos.add(renglon);
        }
        resp.close();
        st.close();
        resultado = datos;
      }
    } catch(SQLException e)  {
      throw(new SQLException());
    }
    return resultado;
  }

  public ArrayList select(String tablas,String campos,String condiciones)throws SQLException{
    String sql="";
    ArrayList resultado = null;
    try  {
      if(condiciones==null || condiciones.equals("")){
        sql = "Select "+campos+" from "+tablas;
      } else{
        sql = "Select "+campos+" from "+tablas+" where "+condiciones;
```

```
        }
        resultado = (ArrayList) sentencia(sql);
      }catch(SQLException e){
        throw(new SQLException());
      }
      return resultado;
    }

    public Integer update(String tablas,String campos,String condiciones)throws SQLException {
      Integer resultado= new Integer(0);
      String sql="";
      try {
        if(condiciones==null || condiciones.equals(""))
          sql="Update "+tablas+" set "+campos;
        else
          sql="Update "+tablas+" set "+campos+" where "+condiciones;
        resultado = (Integer) sentencia(sql);
      } catch(SQLException e)  {
        throw(new SQLException());
      }
      return resultado;
    }


    public Integer delete(String tablas,String condiciones) throws SQLException {
      String sql="";
      Integer resultado= new Integer(0);
      try  {
        if(condiciones==null || condiciones.equals(""))
          sql="DELETE FROM"+tablas;
        else
          sql="DELETE FROM "+tablas+" WHERE "+condiciones;
        resultado = (Integer) sentencia(sql);
      } catch(SQLException e)  {
        throw(new SQLException());
      }
      return resultado;
    }

    public Integer insert(String tabla,String campos,String values) throws SQLException {
      Integer resultado= new Integer(0);
      String sql="";
      try  {
        if(campos.equals("") || campos==null)
          sql="Insert into "+tabla+" values ("+values+")";
        else
          sql="insert into "+tabla+" ("+campos+") values("+values+")";
        resultado = (Integer) sentencia(sql);
      } catch(SQLException e) {
        throw(new SQLException());
      }
      return resultado;
    }
  }
```

## VIII. LeerUsuarioContra.java

```
package tesis.servidor.bd;

import java.io.*;

public class LeerUsuarioContra{

  public String[] obtenDatos() {
    String str="";
    InputStreamReader isr;
    BufferedReader br;
    int cont=0;
    String datos[] = new String[2];
    try {
      FileInputStream archivo= new FileInputStream ("./datos.txt");
      isr = new InputStreamReader(archivo);
      br = new BufferedReader(isr);
      str = br.readLine();
      cont=1;
      int dosPuntos=str.indexOf(":");
      String ident=str.substring(0,dosPuntos).toLowerCase();
      while( str != null )          {
        if(ident.equals("login"))             {
          datos[0]=str.substring(dosPuntos+1,str.length()).trim();
        } else if(ident.equals("password")){
          datos[1]=str.substring(dosPuntos+1,str.length()).trim();
        }
        str = br.readLine();
        if(str!=null){
          dosPuntos=str.indexOf(":");
          ident=str.substring(0,dosPuntos).toLowerCase();
          cont=cont+1;
        }
      }
      br.close();
    }       catch(IOException e){
      e.printStackTrace();
    }
  return datos;
  }
}
```

## IX. Servidor.java

```
package tesis.servidor.rmi;

import java.rmi.Naming;
```

```
public class Servidor{
  public Servidor(){
    try {
      String ip = "localhost.localdomain";
      ConexionRemota c = new ConexionRemotaImpl();
      Naming.rebind("rmi://"+ip+":1099/ServicioConexionRemota", c);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public static void main(String args[]) {
    System.out.println("Iniciando Servidor de Base de Datos");
    new Servidor();
    System.out.println("Servidor de Base de Datos listo para procesar solicitudes ");
  }
}
```

## X. SQLBean.java

```
package tesis;

import java.io.*;

public class SQLBean implements Serializable{

  public static final int INSERT =1;
  public static final int UPDATE =2;
  public static final int DELETE =3;
  public static final int SELECT =4;
  public static final int SENTENCIA =5;
  private int idInstruccion;
  private String sentencia;
  private String tablas;
  private String condicion;
  private String valores;
  private String campos;
  private int tipoSentencia;
  private Object resultado;


  public void setSentencia(String sentencia){
    this.sentencia = sentencia;
  }

  public String getSentencia(){
    if (this.tipoSentencia == SENTENCIA)
      return sentencia;
    else
      return null;
  }
```

```
  public void setTipoSentencia(int tipoSentencia){
    this.tipoSentencia = tipoSentencia;
  }

  public int getTipoSentencia(){
    return tipoSentencia;
  }

  public int getIdInstruccion(){
    return idInstruccion;
  }

  public void setIdInstruccion(int idInstruccion){
    this.idInstruccion = idInstruccion;
  }

  public void setTablas(String tablas){
    this.tablas = tablas;
  }

  public void setCampos(String campos){
    this.campos = campos;
  }

  public void setCondicion(String condicion){
    this.condicion = condicion;
  }

  public String getTablas(){
    return tablas;
  }

  public String getCampos(){
    return campos;
  }

  public String getCondicion(){
    return condicion;
  }

  public void setValores(String valores){
    this.valores = valores;
  }

  public String getValores(){
    return valores;
  }

  public void setResultado(Object resultado){
    this.resultado = resultado;
  }
}
```

## XI. TransaccionDistribuida.java

```
package tesis.cliente.rmi;

import java.sql.*;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
import java.util.*;
import javax.naming.*;
import tesis.SQLBean;

public class TransaccionDistribuida{

 private String nodo;
 private String base;
 private HashMap resultados;
 private HashMap conexionesRemotas;
 private static boolean transaccionExitosa=true;

 private void setTransaccionExitosa(boolean transaccionExitosaA){
  synchronized(this.getClass()){
    if (transaccionExitosa){
      transaccionExitosa = transaccionExitosaA;
    }
  }
 }


 public boolean getTransaccionExitosa(){
  return transaccionExitosa;
 }

 public TransaccionDistribuida(String nodo, String base){
  transaccionExitosa=true;
  this.nodo=nodo;
  this.base=base;
  conexionesRemotas = new HashMap();
  resultados=new HashMap();
 }

 public void setNodo(String nodo){
  this.nodo = nodo;
 }

 public void setBase(String base){
  this.base = base;
 }

 public HashMap getResultados(){
  return resultados;
 }
```

```
public Object getResultado(int idSentencia){
  return resultados.get(new Integer(idSentencia));
}

public boolean agregarSelect(int idSelect, String tabla, String campos, String condiciones){
  SQLBean sqlB = new SQLBean();
  sqlB.setTablas(tabla);
  sqlB.setCampos(campos);
  sqlB.setCondicion(condiciones);
  sqlB.setTipoSentencia(sqlB.SELECT);
  sqlB.setIdInstruccion(idSelect);
  return agregarEnBD(sqlB);
}

public boolean agregarUpdate(int idUpdate, String tabla, String campos, String condiciones){
  SQLBean sqlB = new SQLBean();
  sqlB.setTablas(tabla);
  sqlB.setCampos(campos);
  sqlB.setCondicion(condiciones);
  sqlB.setTipoSentencia(sqlB.UPDATE);
  sqlB.setIdInstruccion(idUpdate);
  return agregarEnBD(sqlB);
}

public boolean agregarInsert(int idInsert, String tabla, String campos, String valores){
  SQLBean sqlB = new SQLBean();
  sqlB.setTablas(tabla);
  sqlB.setCampos(campos);
  sqlB.setValores(valores);
  sqlB.setTipoSentencia(sqlB.INSERT);
  sqlB.setIdInstruccion(idInsert);
  return agregarEnBD(sqlB);
}

public boolean agregarDelete(int idDelete, String tabla, String condiciones){
  SQLBean sqlB = new SQLBean();
  sqlB.setTablas(tabla);
  sqlB.setCondicion(condiciones);
  sqlB.setTipoSentencia(sqlB.DELETE);
  sqlB.setIdInstruccion(idDelete);
  return agregarEnBD(sqlB);
}

public boolean agregarSentencia(int idSentencia, String sentencia){
  SQLBean sqlB = new SQLBean();
  sqlB.setTipoSentencia(sqlB.SENTENCIA);
  sqlB.setSentencia(sentencia);
  sqlB.setIdInstruccion(idSentencia);
  return agregarEnBD(sqlB);
}

private boolean agregarEnBD(SQLBean sentencia){
  if (conexionesRemotas.containsKey(nodo)){
    HashMap basesDeNodo = (HashMap) conexionesRemotas.get(nodo);
    if (basesDeNodo.containsKey(base)){
```

```
        Collection sentenciasSQL = (ArrayList) basesDeNodo.get(base);
        sentenciasSQL.add(sentencia);
      } else {
        Collection sentenciasSQL = new ArrayList();
        sentenciasSQL.add(sentencia);
        basesDeNodo.put(base, sentenciasSQL);
      }
    } else {
      HashMap basesDeNodo = new HashMap();
      Collection sentenciasSQL = new ArrayList();
      sentenciasSQL.add(sentencia);
      basesDeNodo.put(base, sentenciasSQL);
      conexionesRemotas.put(nodo,basesDeNodo);
    }
    return true;
  }

  public boolean ejecutarTransaccion(){
    Set nodos = conexionesRemotas.keySet();
    Iterator it = nodos.iterator();
    boolean transaccionExitosa = false;
    HashMap resultadoParcial = null;
    int indexHilo = 0;
    GeneraHiloConexionRemota.setHilosLanzados(conexionesRemotas.size());
    try {
      while (it.hasNext()){
        ++indexHilo;
        resultadoParcial = null;
        String ip = (String) it.next();
        HashMap bases = (HashMap) conexionesRemotas.get(ip);
        GeneraHiloConexionRemota hcr= new GeneraHiloConexionRemota(ip,bases);
        Thread hiloActual = new Thread(hcr,"H("+ip+")");
        hiloActual.start();
      }
    } catch (Exception murle) {
      murle.printStackTrace();
    } finally {
      while (!GeneraHiloConexionRemota.getHilosTerminados()){
    }
    this.resultados=GeneraHiloConexionRemota.getResultadoTotal();
    GeneraHiloConexionRemota.inicializaBanderas();
  }
  return transaccionExitosa;
  }
}
```