5
2ej

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

Facultad de Contaduría y Administración

PROTOTIPO DE UNA APLICACION DE TARJETA DE CREDITO UTILIZANDO EL MODELO C/S

SEMINARIO DE INVESTIGACION INFORMATICA

Que para obtener el título de:

LICENCIADO EN INFORMATICA

presentan

ISRAEL GONZALEZ MIJANGOS

HECTOR RENATO LAZO MARTINEZ

MARCO ANTONIO MATURANO SALERO

Asesor del Seminario: Ing. Miguel Santiago Suárez Castañón

México, D. F. 1999

275324

A nuestros padres, hermanos y amigos por su apoyo total.

A los profesores que con su profesionalismo y capacidad me inculcaron un espíritu de superación.

INTRODUCCION

Durante la década de los 90's la evolución tecnológica de los sistemas de información ha sido vertiginosa. Dicho avance de la tecnología se ha hecho presente en la continua mejora de los equipos de cómputo para procesar los datos, hasta la forma de diseñar una aplicación.

Las empresas han sufrido un impacto directo de este avance, ya que la competencia se toma mucho más agresiva y, por lo tanto, la necesidad de tener la información adecuada en el momento oportuno se toma de suma importancia para el desarrollo y éxito de la empresa. Este último punto tiene como consecuencia que las empresas y organizaciones hagan una continua mejora de sus servicios, con lo cual se pretende beneficiar al cliente y, por ende obligar a las empresas a contar con mejores sistemas de información, con el menor impacto posible en los costos.

Con base en el antecendente anterior, varias alternativas de tecnología se han ido desarrollando, de manera que permitan a las empresas y organizaciones implantar o mejorar los sistemas de información con que cuentan. Sin embargo, una empresa debe de optar por la tecnología que no sólo le permita desarrollarse, sino que también represente una solución en el futuro. Es decir, la tecnología seleccionada no sólo tendrá que satisfacer las necesidades actuales, sino también las futuras.

El manejo transaccional de la información cobra cada vez más fuerza, especialmente si se considera que la mayoría de los datos que consolidan la información, se pueden encontrar distribuidos en varios puntos de operación del negocio. Ligado a esto, los requerimientos de información en línea y la interoperabilidad de diversas aplicaciones y ambientes son primordiales para tener información confiable.

Típicamente, las empresas utilizan aplicaciones de negocio críticas, que actualmente cuentan entre sus desventajas:

- Lentitud en la manipulación de información dentro de las aplicaciones.
- Proceso de información por lotes (Con la necesidad de hacerlo en línea).
- Carencia de administración de las aplicaciones.
- Falta de integración entre diversas plataformas.
- Convivencia con distintos manejadores de bases de datos.

Estos, solo por mencionar algunos de los muchos problemas a los que se enfrentan diariamente en la operación de dichos sistemas.

A partir de este escenario, resulta de interés presentar una tesis que dedique especial atención en una solución innovadora, profunda y profesional, apoyada con tecnología de punta, que permita contar con una opción para resolver los problemas que las empresas con sistemas de información crítica, operan en la actualidad.

El objetivo de la presente tesis es, presentar el prototipo de una aplicación, que mediante el uso de la arquitectura Cliente/Servidor y un monitor de transacciones ofrezca una visión de la potencialidad del uso de la combinación de la tecnología mencionada, buscando el beneficio al lograr implantar mejores sistemas de información críticos.

A lo largo de los capítulos que conforman esta tesis, se pretende explorar los conceptos y elementos necesarios para poder implantar un sistema de 3 capas de Cliente/Servidor. En algunos capítulos, se hace mención a nuevas fuentes de tecnología, así como sus opciones; sin embargo, no es objetivo de esta tesis explicar dichos elementos, pero han sido incluidos para poder obtener un contexto general de la tecnología Cliente/Servidor.

En el Capitulo 1 se establece el marco teórico de la arquitectura o modelo Cliente/Servidor, además de dar los elementos básicos necesarios para entender el uso de dicha tecnología.

En el Capitulo 2 se introducen los conceptos referentes a lo que es un monitor de transacciones. Así como los conceptos que involucra dicho elemento.

En el Capitulo 3 se presentan los distintos paradigmas de comunicación, con los cuales puede ser implantada una aplicación distribuida.

En el Capitulo 4 se describe la arquitectura del prototipo y las características funcionales de este. Se mencionan varias posibilidades de expansión de funcionalidad para implantar un sistema real.

En el Capitulo 5 se describe el proceso de desarrollo del Prototipo de Tarjeta de Crédito.

Finalmente, se presentan las conclusiones que se obtuvieron durante el desarrollo este trabajo.

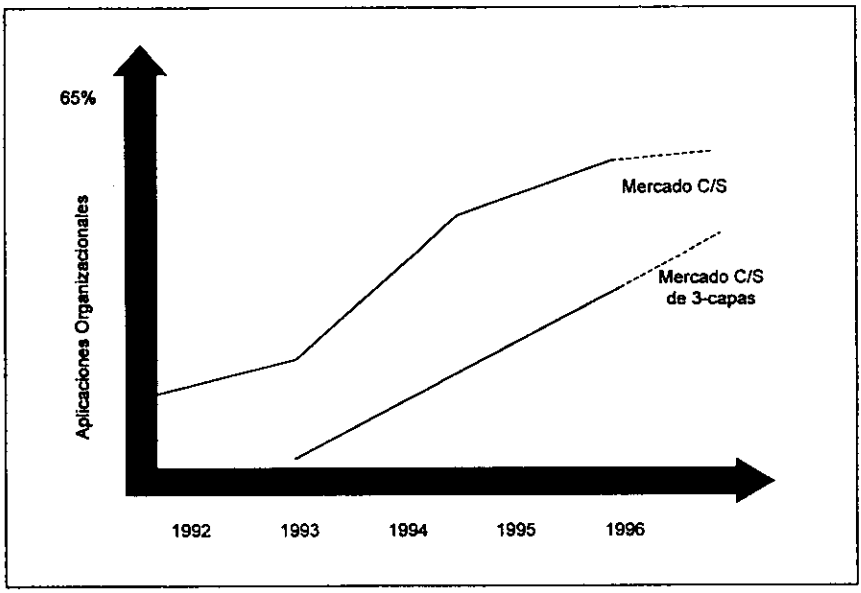
Es importante mencionar que la presente tesis, tiene un alto grado de aplicación en el sector de sistemas de aplicaciones críticas, calificando para este nicho, organizaciones de carácter financiero, gubernamental, y en general todas aquellas que requieran del proceso de un gran volumen de transacciones y alta confiabilidad.

Capitulo 1

El objetivo de este capítulo es establecer el marco teórico de la arquitectura o modelo Cliente/Servidor, además de dar los elementos básicos necesarios para entender el uso de dicha tecnología.

Introducción a Cliente/Servidor

Años después de su introducción, Cliente/Servidor (referido como C/S de ahora en adelante) se ha convertido en la arquitectura de aplicaciones por selección. C/S fue el verdugo de las aplicaciones monolíticas de mainframe para dividir la carga de proceso entre clientes y servidores. Como resultado, C/S ha revolucionado la forma en que diseñamos y construimos aplicaciones. Además añadió expectativas acerca del 'look and feel' del software para usuarios.



1.1 2-capas vs 3-capas (Fuente: Standish Group, 1996 COMPASS Survey)

En este despertar, C/S originó una industria de software dominada por gigantes como Baan, Informix, Oracle, Lotus, Microsoft, Novell, SAP, Sun, Sybase y muchos otros. Estas compañías son las estrellas de la primera era de C/S, y son ahora la alternativa de soluciones en el mercado.

Sin embargo, dentro de la revolución de C/S, hay una revolución interna. C/S esta transformando rápidamente su arquitectura de 2-capas a 3-capas. El impacto de este cambio es mayor que cuando se desplazaron las aplicaciones monolíticas hacia las aplicaciones C/S. El movimiento a una arquitectura de 3-capas nació con la necesidad de hacer que C/S trabajara con aplicaciones empresariales de alta demanda. Sin embargo, internet, objetos distribuidos y componentes, son ahora los que impulsan la arquitectura C/S de 3-capas.

C/S se ha convertido en el modelo dominante del mundo de aplicaciones empresariales de alta demanda y esto, esta respaldado por la arquitectura de 3-capas.

El desarrollo con C/S requiere de habilidades distintas como son el manejo y proceso de transacciones, diseño de base de datos, experiencia en comunicaciones y además conocimiento de interfaces gráficas. Las aplicaciones más avanzadas requieren de conocimiento de objetos distribuidos e internet.

¿Qué es Cliente/Servidor?

A pesar de ser una tecnología de vanguardia no hay un termino que defina a C/S. El nombre implica clientes y servidores, los cuales son entidades lógicas separadas que trabajan en conjunto a través de una red para lograr un objetivo especifico. ¿Qué es lo que hace a C/S distinto de otros tipos de software distribuido? Un sistema cliente servidor cuenta con los siguientes elementos:

Servicio: C/S es una relación entre procesos ejecutándose en máquinas separadas. La función del servidor es proveer de servicios; el cliente es un consumidor de esos servicios. En esencia, C/S provee una clara separación de funciones con base en la idea de servicios.

Recursos Compartidos: Un servidor puede servir a muchos clientes al mismo tiempo y regular su acceso a los recursos compartidos.

Servicio: C/S es una relación entre procesos que se ejecutan en máquinas separadas. El proceso servidor es un proveedor de servicios. El cliente es un consumidor de servicios. En esencia, C/S provee una separación limpia de funciones, con base en la idea de servicios.

Protocolos Asimétricos: Hay una relación de muchos a uno entre clientes y servidores. Los clientes siempre inician un dialogo requiriendo un servicio. Los servidores esperan de manera pasiva las peticiones de los clientes.

Transparencia de Locación: El servidor es un proceso que puede residir en la misma máquina que el cliente o en una máquina diferente en una red. El software de C/S generalmente 'encubre' la locación de los servidores a los clientes al redireccionar las peticiones cuando es necesario. Un programa puede ser un cliente, un servidor o ambos.

Combinar y Ajustar: El software ideal en C/S es independiente del hardware o del sistema operativo. Se debe tener la posibilidad de poder combinar y ajustar las plataformas de clientes y servidores.

Intercambio de mensajes: Los clientes y servidores son un par de sistemas que están acoplados y que interactúan con base en un mecanismo de mensajes. El mensaje es el mecanismo de entrega para las peticiones de los servicios y respuestas.

Encapsulamiento de servicios: El servidor es un 'especialista'. Un mensaje indica a un servidor que servicio se requiere; corresponde entonces al servidor determinar cómo hacer el trabajo. Los servidores pueden ser modificados sin afectar los clientes mientras la estructura del mensaje no sea modificada.

Escalabilidad: Los sistemas C/S pueden ser escalados horizontalmente y verticalmente. Escalar de manera horizontal significa añadir o quitar clientes workstation con un impacto pequeño en el desempeño. Escalar de manera vertical significa migrar a un servidor con más capacidad de proceso o múltiples servidores.

Integridad: El código de servidor y los datos son mantenidos centralmente, lo cual resulta en un mantenimiento más barato y el cuidado de la integridad de los datos compartidos. Al mismo tiempo, los clientes permanecen personales e independientes.

Muchos sistemas con distintas arquitecturas han sido llamados 'cliente/servidor'. Muchos vendedores de sistemas aplican el término a sus paquetes. Por ejemplo, los vendedores de servidores de archivos claman haber inventado el término, y los vendedores de servidores de bases de datos son conocidos en algunos círculos como los vendedores de cliente/servidor. La idea de dividir una aplicación en clientes y servidores ha sido utilizada por más de diez años para crear varias formas de software para redes de área local.

Generalmente estas soluciones están hechas a la medida y muchas son vendidas por más de un vendedor. Cada una de estas soluciones, sin embargo, se distingue por la naturaleza de los servicios que ofrece a los clientes, algunas de ellas son las siguientes:

Servidores de Archivos – En este esquema, el cliente (típicamente una PC) pasa peticiones sobre registros de archivos por una red a un servidor. Esta forma requiere de un intercambio de mensajes grande por la red para localizar el archivo adecuado.

Servidores de base de datos – El cliente pasa peticiones de SQL como un mensaje al servidor de la base de datos. Los resultados de cada comando SQL son regresados por la red. El código que procesa la petición de SQL y los datos, residen en la misma máquina. El servidor utiliza su capacidad de procesamiento propia para proporcionar al cliente los datos requeridos en lugar de todos los datos, como sucede en el servidor de archivos. El resultado es un manejo más eficiente del proceso distribuido. Con este enfoque, el código del servidor es hecho por el vendedor. Pero generalmente, el código que requieren los clientes, necesita ser desarrollado.

Servidores de Transacciones – El cliente invoca procedimientos remotos que residen en un servidor con una maquinaria de SQL integrada. Estos procedimientos remotos en el servidor, ejecutan un grupo de sentencias SQL. El intercambio en la red, consiste de un mensaje, una petición/respuesta (al contrario de un servidor de base de datos donde existe una petición/respuesta por cada sentencia SQL en una transacción). Las sentencias SQL, todas son exitosas o fallan como una unidad. Estas sentencias SQL agrupadas son denominadas transacciones. Con un servidor de transacciones, se pueden crear aplicaciones C/S al escribir los componentes servidor y cliente. El cliente generalmente involucra una interfase gráfica. El servidor generalmente se compone de sentencias SQL, que se ejecutaran en una base de datos. Estas aplicaciones son llamadas de Procesamiento de Transacciones en Línea (OLTP – Online Transaction Processing) u OLTP. Generalmente son aplicaciones consideradas como críticas que

requieren de 1-3 segundos de respuesta el 100% del tiempo. Las aplicaciones OLTP también requieren gran control en los aspectos de seguridad e integridad de la base de datos.

Servidores de Objetos – Una aplicación C/S es escrita, en este modelo, como un grupo de objetos que se comunican. Los objetos clientes se comunican con los objetos servidores utilizando un componente conocido como Object Request Broker (ORB). El cliente invoca un método en un objeto remoto. El ORB localiza la instancia del objeto en cuestión, invoca el método requerido, y regresa los resultados al objeto cliente. Los servidores de objetos deben de proveer soporte para acceso concurrente y recursos compartidos.

Servidores de Web – Este es uno de los ejemplos de aplicaciones C/S a un nivel considerado como el mejor modelo. En este, un cliente delgado, portáble y universal habla con un servidor gordo. En el modelo más simple, un servidor web regresa documentos que un cliente requiere por nombre. Los clientes y los servidores se comunican utilizando comunicación por RPC (Remote Procedure Call), en un protocolo denominado HTTP (Hyper Text Transfer Protocol). Este protocolo define un conjunto de instrucciones sencillas, los parámetros son pasados como cadenas, sin indicar los tipos de datos.

Concepto de 'Capas'

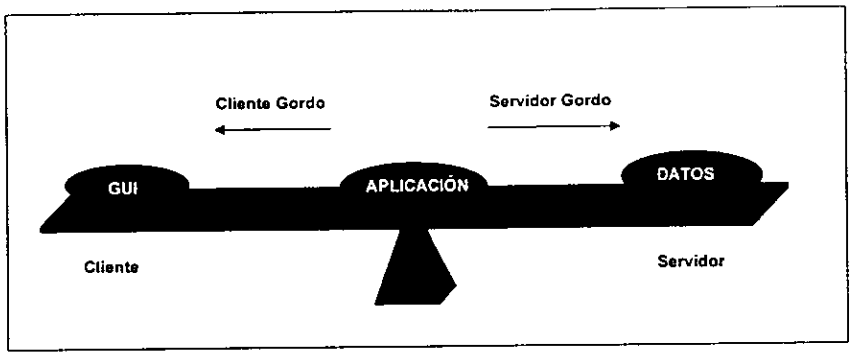
En los años 80's, los vendedores de minicomputadoras introdujeron el término 3-capas (como en 'arquitectura de 3-capas') para describir la partición física de una aplicación entre terminales (capa 1), minicomputadoras (capa 2), y mainframes (capa 3). Esto les permitió vender las computadoras de rango medio como front-end para los mainframes.

Hoy, se utiliza el término de capas para describir el particionamiento lógico de una aplicación entre clientes y servidores. El dividir la carga de proceso es un concepto elemental de C/S. Pero además introduce innecesariamente y de manera previa – ahora persistente – el diseño de donde colocar esta carga. Las capas nos permiten describir las opciones arquitectónicas básicas:

2-capas divide la carga de proceso en dos. La mayoría de la lógica de la aplicación se ejecuta en el cliente, el cual, típicamente envía peticiones de SQL a un servidor residente en la base de datos. Esta arquitectura se denomina de 'cliente gordo', ya que una gran parte de la aplicación se ejecuta en el cliente.

3-capas divide la carga de proceso entre 1) los clientes ejecutando la lógica de interfase gráfica (GUI), 2) el servidor aplicativo ejecutando la lógica del negocio y 3) la base de datos y/o aplicación propietaria. Ya que 3-capas mueve la lógica de la aplicación al servidor, también se conoce como arquitectura de 'servidor gordo' – o recientemente como 'cliente-delgado'.

Por definición, todas las aplicaciones C/S deben de tener al menos dos capas: La interfase de usuario que reside en el cliente y los datos compartidos que residen en los servidores. Como se puede ver, una aplicación es 2-capas o 3-capas con base en la separación de la lógica de la aplicación del GUI y la base de datos.



1.2 Cliente gordo vs Servidor Gordo

Este particionamiento es un punto de diseño importante que hace una gran diferencia en determinar el éxito de aplicaciones de misión-crítica. Hoy, los diseñadores de aplicaciones y desarrolladores tienen errores arquitectónicos que cuestan millones de dólares en conjunto. Las aplicaciones generalmente fallan debido a estos errores –no por problemas de codificación. Aunque los últimos se pueden detectar y corregir, ***un proyecto difícilmente se puede recuperar de un error arquitectónico***. Las buenas noticias son que estos errores arquitectónicos se pueden evitar con un entendimiento básico de las ventajas y desventajas de las arquitecturas de C/S.

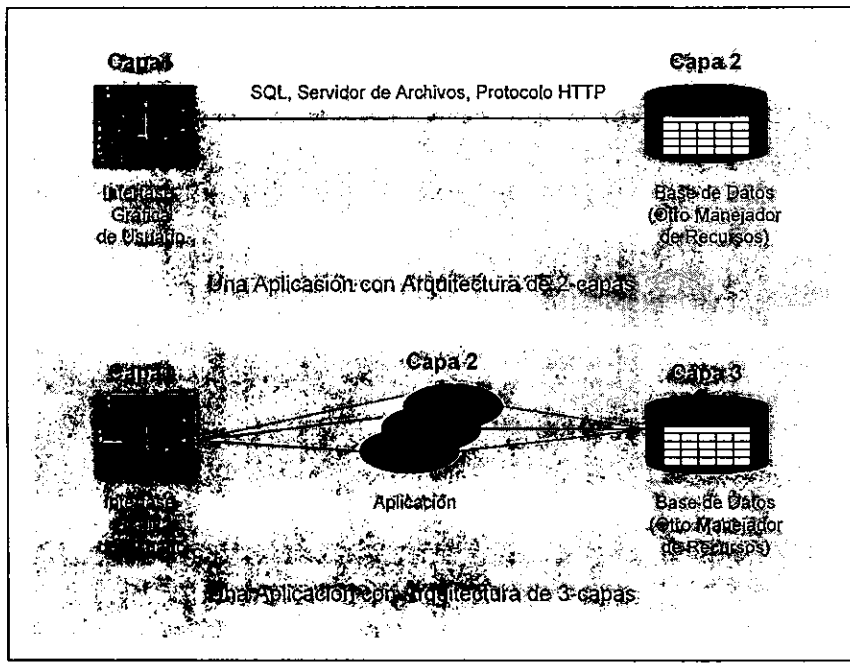
Beneficios y Límites de la Arquitectura de 2-capas

En sistemas C/S de 2-capas, la lógica de la aplicación se incluye en la interfase de usuario en el cliente o dentro de la base de datos en el servidor (en algunos casos en ambos). Se ejecuta un cliente con GUI. Envía una sentencia SQL, llamadas al sistema de archivos, o comandos HTTP por la red hacia el servidor. El servidor procesa la petición y regresa un resultado o un conjunto de datos. Para accesar los datos, los clientes tienen que saber como están organizados y almacenados en el servidor. Una variación de enfoque de 2-capas utiliza *stored procedures* para compartir el proceso con el lado del servidor. En lugar de enviar peticiones de SQL a través de la red, los stored procedures permiten invocar a una función que se ejecuta internamente en la base de datos – se puede considerar este particionamiento como 2.5-capas.

La simplicidad es el factor más grande que está incluido en la popularidad de C/S de 2-capas. Este es útil cuando ser crean aplicaciones de manera rápida mediante el apoyo de herramientas de construcción visuales. Típicamente estas aplicaciones son de tipo departamental, como soporte de decisiones, y software para grupo de trabajos, o aplicaciones de web sencillas.

Como eran exitosas estas aplicaciones, se comenzaron a hacer populares, de pronto los arquitectos de aplicaciones se vieron dependientes de la arquitectura C/S de 2-capas. Muy pronto averiguaron, que a pesar del éxito de la aplicación, la arquitectura misma de la aplicación y las herramientas no podían ser escaladas. Las aplicaciones que trabajaban de manera perfecta en prototipos y pequeñas instalaciones, no resistieron el peso de un ambiente de producción masivo. El modelo C/S de 2-capas no esta diseñado para misión-crítica. Como consecuencia las fechas de liberación de dichas aplicaciones fueron retrasadas, los costos de los proyectos se elevaron como consecuencia de buscar soluciones al problema. El software propietario encontró problemas arquitectónicos cuando se intentaba escalar la aplicación. Naturalmente, el fracaso de C/S como modelo de desarrollo fue declarado.

Lo que realmente sucedió, es que las aplicaciones departamentales comenzaron a convertirse en aplicaciones críticas, y por lo tanto C/S sufrió una transición. Actualmente se implantan aplicaciones C/S de misión-crítica y de comercio electrónico que utilizan C/S de 3-capas. El movimiento de 2-capas a 3-capas se dio por la necesidad de información, y en este mundo de información, las aplicaciones ahora están integradas de componentes y distribuidas entre varios procesadores en un mundo de 3-capas.



1.3 2-capas vs 3-capas

C/S de 3-capas

C/S ha rebasado los supuestos con los cuales fue creado. Las aplicaciones creadas ahora con este modelo atienden a miles de usuarios con una aplicación de misión-crítica por ejemplo. Estas aplicaciones generalmente se ejecutan en varios servidores y consisten de cientos de componentes de software.

En el mundo actual las transacciones pueden provenir de los consumidores, los distribuidores, los proveedores, de los empleados de una misma compañía. C/S permite a las empresas sobrevivir en un cambiante clima de negocios donde son presionadas por nuevas demandas que deben de ser atendidas inmediatamente. Ejemplos:

- Las empresas compiten agresivamente por ser los primeros en llegar al mercado con nuevos productos o servicios. El éxito que tienen llega a depender en gran parte de las aplicaciones con las que cuentan.

- Las compañías crean corporaciones virtuales a través de alianzas con un buen respaldo de aliados tecnológicos. Esto les permite reaccionar de mejor manera a las nuevas oportunidades – y mantener un buen seguimiento a sus competidores.

- Los roles y las relaciones entre las empresas son cambiantes de acuerdo al ritmo de la industria. Las compañías exitosas visualizan dichos cambios para tener la oportunidad de incrementar su participación en el mercado y para adquirir una posición dominante en la industria. Las fusiones y adquisiciones crecen.

En su papel, C/S permite y maneja dichos cambios. Cambia el modo en que muchas empresas operan. Pero no se aprovecha al máximo debido a limitantes tecnológicas – incluyendo redes de bajo costo y con

gran ancho de banda, una nueva generación de sistemas de escritorio habilitados para comunicación en red, e infraestructura distribuida orientada a componentes.

En la arquitectura de 3-capas, el cliente provee el GUI e interactua con el servidor a través de un servicio remoto o invocación de métodos. La lógica de la aplicación vive en la capa media. Se implanta de forma separada de la interfase del cliente y la base de datos. Así la lógica se ejecuta ahora en su propia capa y puede ejecutarse en uno o varios servidores.

3-capas es el área de crecimiento de C/S para cómputo masivo, ya que llena los requerimientos de aplicaciones de gran escala, incluyendo aquellas de internet. Las aplicaciones de 3-capas son más fáciles de administrar e implantar en una red – la mayoría del código se ejecuta en el servidor. Además, dichas aplicaciones minimizan los intercambios de información en la red al crear niveles abstractos de servicio. En lugar de interactuar con la base de datos de manera directa, el cliente llama la lógica en los servidores. La lógica del negocio es quien accesa entonces la base de datos en nombre del cliente. Las 3-capas sustituyen unas pocas llamadas de servidores por muchas sentencias SQL, así que el rendimiento es mejor que en 2-capas. También provee de una mayor seguridad al no exponer el esquema de la base de datos al cliente y mediante la habilitación de una autorización, más granular en el servidor.

Comparación entre 2-capas y 3-capas

La siguiente tabla muestra las comparaciones entre los enfoques de 2 y 3 capas. Cuando C/S era departamental las limitantes de 2-capas no eran de mayor importancia. Sin embargo conforme las aplicaciones se fueron convirtiendo en aplicaciones de misión-crítica, el enfoque de 3-capas se hizo esencial.

Modelo de 2-capas vs. Modelo de 3-capas

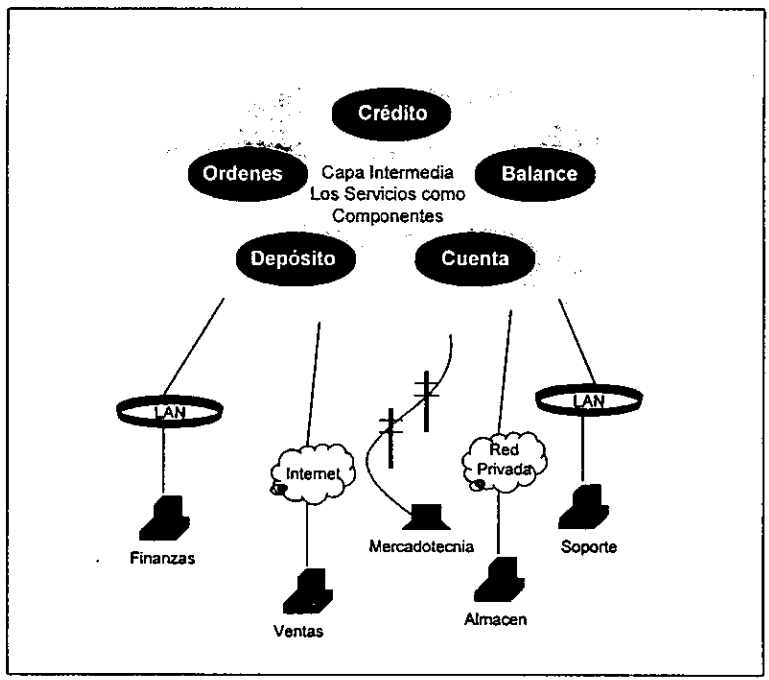
Administración del Sistema	Compleja (Mayor lógica en el cliente que manejar)	Menos compleja (La aplicación puede ser administrada de manera centralizada en el servidor – los programas de la aplicación son hechos visibles a las herramientas de admon.)
Seguridad	Poca (Al nivel de datos)	Alta (Al nivel de servicio o método)
Encapsulación de datos	Poca (Las tablas de datos están expuestas)	Alta (El cliente invoca servicios o métodos)
Rendimiento	Pobre (Muchas sentencias SQL son enviadas a través de la red; los datos seleccionados son regresados al cliente para su análisis)	Bueno (Solo las peticiones a los servicios y las respuestas son enviadas entre el cliente y el servidor)
Escalabilidad	Pobre (Limitación en los enlaces de comunicación de los clientes)	Excelente (Concentra sesiones nuevas; y puede distribuir carga entre multiples servidores.)
Reuso de Aplicación	Pobre (La aplicación es monolítica y es generalmente el cliente)	Excelente (Puede reutilizar los servicios y objetos de la aplicación)

Facilidad de Desarrollo	**Alta**	Mejorando (Las herramientas estándar pueden ser utilizadas para crear los clientes, y nuevas herramientas estan apareciendo que sirven para generar tanto los clientes como los servidores)
Infraestrutura de Servidor a Servidor	No	Si (Via middleware)
Integración con Aplicaciones Propietarias	No	Si (Via gateways encapsulados por los servicios u objetos)
Soporte a Internet	Pobre (Las limitaciones en el ancho de banda de internet hacen difícil de obtener los clientes gordos)	Excelente (Los clientes delgados son fáciles de obtener de la red (applets o beans), y el proceso distribuido distribuye la carga a los servidores)
Soporte a bases de datos heterogeneas	No	Si (Las aplicaciones de 3-capas pueden utilizar multiples bases de datos dentro de una misma transacción)
Alterativas de comunicación	No (Solamente el modo sincrono, o llamadas tipo RPC)	Si (Soporta llamadas tipo RPC (Sincronas), colas, conversacion, asincrono, etc)
Flexibilidad de Arquitectura de Hardware	Limitada (Se tiene un cliente y un servidor)	Excelente (Las 3-capas pueden estar representadas por distinto hardware, la segunda y tercera capa pueden coexistir en una máquina, o la segunda capa puede estar conformada de distintas máquinas o ambientes de harware)

Disponibilidad	Pobre (No hay nodo de respaldo)	Excelente (Puede reiniciar los componentes de la 2-capa en otros equipos de hardware)

Componentes: Cuando 3-capas son N-capas

La capa intermedia en la mayoría de las aplicaciones de 3-capas no esta implantada como un programa monolítico. Esta implantada como una colección de componentes que son utilizados en una variedad de transacciones del negocio iniciadas por el cliente.



1.4 Componentes de una Arquitectura de N-capas

Cada componente automatiza una pequeña función del negocio. Los clientes frecuentemente combinan muchos componentes de la capa intermedia dentro de una transacción del negocio. Un componente puede llamar a otros componentes para ayudar a implantar una petición. Además algunos componentes pueden actuar como gateways que encapsulan aplicaciones propietarias ejecutándose en mainframes. Así que, la mayoría del tiempo, la arquitectura de 3-capas, es en realidad de N-capas.

Beneficios de una Arquitectura basada en Componentes.

Las aplicaciones basadas en componentes ofrecen ventajas significativas sobre las aplicaciones monolíticas. Cuando se diseña la capa intermedia como una aplicación de componentes, se obtienen los siguientes beneficios:

Desarrollo de grandes aplicaciones en pasos pequeños. La arquitectura basada en componentes permite desarrollar aplicaciones de misión-crítica de gran escala como pequeños proyectos. Cuando se utiliza este método de desarrollo, se puede tener versiones iniciales de la aplicación en producción de manera más rápida. También reduce el riesgo. Standish Group reporta que entre más crezca un proyecto, más probabilidades de fracasar tendrá. Encontraron que el 53% de los proyectos fallan. Los proyectos pequeños que son desarrollados por equipos de 4 personas por cuatro meses tienen mejores oportunidades de éxito. La filosofía de componentes se adapta bien a esta filosofía de proyectos pequeños.

Aplicaciones que reusan componentes. A diferencia de los lenguajes orientados a objetos que se enfocan en el reuso de código, las aplicaciones reusan los componentes como cajas negras de objetos binarios. Se pueden combinar en diferentes maneras, dependiendo de la aplicación.

Los clientes accesan los datos y las funciones de manera segura y fácil. Los clientes envían sus peticiones a los componentes para que ejecuten una función en su nombre. Los componentes del servidor encapsulan los detalles de la lógica de la aplicación y así aumentan el nivel de abstracción. Los clientes no necesitan saber a que base de datos se esta accesando para ejecutar la petición. Y no necesitan saber si la petición fue enviada a otro componente o aplicación para su ejecución. La encapsulación provee un acceso consistente, seguro y auditable, además de eliminar actualizaciones aleatorias sin control provenientes de otras aplicaciones en determinado momento.

Las aplicaciones a la medida pueden incluir componentes de terceros. Las compañías obtienen grandes beneficios cuando integran dentro de sus aplicaciones componentes desarrollados por terceros. Los beneficios que se obtienen de incluir estos componentes generalmente son funcionalidad particular probada y lista para integrarse sin mayores problemas de funcionamiento.

Los ambientes de componentes no envejecen, solo se mejoran. Cuando una aplicación es construida con componentes, se fija la base para poder crecer esa aplicación a un conjunto de aplicaciones que se integren de manera funcional. Se pueden añadir nuevos clientes, nuevos servicios, se pueden cambiar o actualizar algunos componente de acuerdo a las necesidades que la empresa vaya teniendo, de tal manera que el sistema siempre se encuentre actualizado para un funcionamiento óptimo.

Componentes del Tipo Servidor

Hay dos tipos de componentes en la capa intermedia:

Servicios que implantan alguna función del negocio – por ejemplo, consulta de saldo. Los servicios son procedimientos que no tienen estado: cuando reciben una petición, la lógica de la aplicación debe accesar y actualizar los datos. La mayoría del middleware actual soporta servicios procedurales.

Objetos que exponen un conjunto de procedimientos relacionados o métodos, no solo un procedimiento. La infraestructura maneja todos los métodos relacionados con una actualización, una consulta, una auditoría o una eliminación de determinada información como una unidad. Hoy, los ORB's (Object Request Broker por sus siglas en inglés) proveen de una infraestructura de objetos distribuidos. Estos ORB's permiten a los objetos comunicarse entre lenguajes, sistemas operativos, y redes. Dependiendo en su implantación, los objetos pueden ser componentes sin estado o con estado.

Objetos sin estado no tienen un estado único. Cuando un objeto sin estado es invocado, debe determinar que instancia de datos necesita y luego los recupera de la base de datos. Cuando ha terminado, debe actualizar la base de datos. DCOM de Microsoft (Distributed Component Object Model por sus siglas en inglés) es un ejemplo de un ambiente de objetos sin estado. A pesar de que el estándar de ORB, CORBA (Common Object Request Broker Architecture por sus siglas en inglés), soporta objetos con y sin estado, la mayoría de los ORB's son sin estado.

Objetos con estado habilitan a los clientes para realizar peticiones a un objeto específico utilizando un identificador único. La capa intermedia de software debe de entregar la petición al objeto en particular. Si no esta en memoria ya, la infraestructura debe de encontrar dicho objeto y cargar el estado del objeto y sus métodos. Después de que la petición ha sido atendida, la infraestructura debe de guardar (o hacer commit) el estado y borrar (o hacer colección de basura) el objeto de la memoria.

Como se Comunican los Componentes entre sí

El cliente envía peticiones a los componentes utilizando nombre lógicos en lugar de direcciones físicas. La infraestructura de la capa intermedia mapea estos nombres lógicos a locaciones físicas y asegura la entrega del mensaje. Este mapeo provee transparencia de locación para los componentes servidor. Obviamente, esto facilita la vida para los clientes que buscan por algún componente en particular. Permite a la infraestructura de la capa intermedia manejar réplicas de los componentes para mantener la seguridad de la aplicación libre de fallas en el sistema. La infraestructura también utiliza las réplicas para manejar cargas de trabajo intensas y balancear dicha carga. Así la infraestructura en la capa intermedia dirige la petición a la locación física donde se esté ejecutando el componente.

Naturalmente el modelo de programación de la capa intermedia determina si llama servicios u objetos. Hoy los monitores de transacciones están orientados a servicios, mientras que los ORB's están orientados hacia objetos. Sin embargo una tendencia es que ambos se integren para dar paso a los OTM's (Object Transaction Monitor por sus siglas en inglés).

La capa intermedia debe de proporcionar cualesquiera de las alternativas de comunicación siguientes, sino es que todas ellas:

Conversaciones, que soportan un diálogo continuo involucrando muchas transacciones entre clientes y componentes de servidor. Ejemplos de esto, los sockets de TCP/IP.

Petición – Respuesta (Sincrona), que soporta una sola interacción entre el cliente y el (los) componente(s) de servidor. Ejemplos, los RPC's (Remote Procedure Calls por sus siglas en inglés).

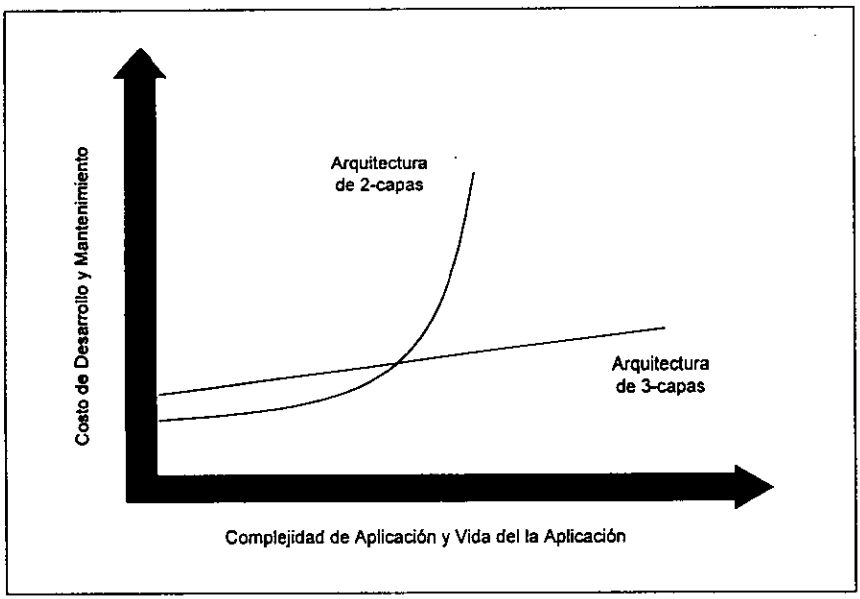
Colas, donde la interacción entre el cliente y el componente de servidor es deshabilitada. Los mensajes son encolados para los servidores. Los servidores accesan dichos mensajes cuando están listos. Las colas pueden soportar mensajes con distintas prioridades. Soportan activación por medio de mecanismos de tiempo. Las colas son necesarias muchas veces en una arquitectura de 3-capas.

Publicación y Suscripción, que habilita a los clientes (o componentes de servidor) a registrarse con el fin de enterarse de determinados mensajes manejados por un manejador de eventos. Los Servidores o componentes (o clientes) publican los mensajes al manejador de eventos. El manejador de eventos actúa como un despachador, enviando los mensajes publicados a los suscriptores que han declarado su interés por recibir dichos mensajes.

Notificación y Datagramas que permiten la comunicación en un solo sentido a uno o más componentes o clientes.

Cuando usar la Arquitectura de 3-capas

El modelo de 3-capas crece rápidamente, sin embargo el modelo de 2-capas no esta en desuso. Hay todavía aplicaciones que pueden tomar ventaja de lo que ofrece el modelo de 2-capas. La pregunta radica en cuando usar 2-capas o 3-capas. Gartner Group sugiere tomar en cuenta los siguientes puntos:



1.1 2-capas y 3-capas comparados (Fuente: Gartner Group)

- Muchos servicios aplicativos o clases – más de 50.
- Las aplicaciones están programadas en distintos lenguajes o escritas por distintas organizaciones.

- Dos o más fuentes de datos heterogéneas – como por ejemplo dos DBMS's o un DBMS y sistema de archivos.
- Donde el ciclo de vida de una aplicación es mayor a tres años – especialmente si se esperan muchas modificaciones o adiciones.
- Un alto volumen de trabajo – más de 50000 transacciones por día o más de 300 usuarios concurrentes en el mismo sistema accesando la misma base de datos.
- La proyección de que la aplicación crecerá con el tiempo de manera que alguno de los puntos anteriores aplique.

Es importante mencionar que el modelo de 2-capas aplica cuando son proyectos pequeños, ya que es fácil diseñar en dos capas la arquitectura de una aplicación así, sin embargo en aplicaciones grandes, es importante considerar el modelo de 3-capas por las dificultades de manejo que representa por su tamaño.

Capitulo 2

El objetivo de este capítulo es introducir los conceptos referentes a lo que es un monitor de transacciones. Así como los conceptos que involucra dicho elemento.

Monitores de Transacciones, corazón del Modelo de 3-capas

Los Monitores de Proceso de Transacciones (TP Monitors) están siendo redescubiertos por una nueva generación de arquitectos de software. De acuerdo a Standish Group, los monitores de transacciones llegaron a ser la tecnología de mayor uso en 1996 – El 57% de las aplicaciones de misión-crítica fueron construidas utilizando algún monitor de transacciones. Este uso de los monitores se debe a que juegan un papel fundamental en el modelo de 3-capas de C/S. Proveen del esquema para ejecutar procesos en la capa intermedia.

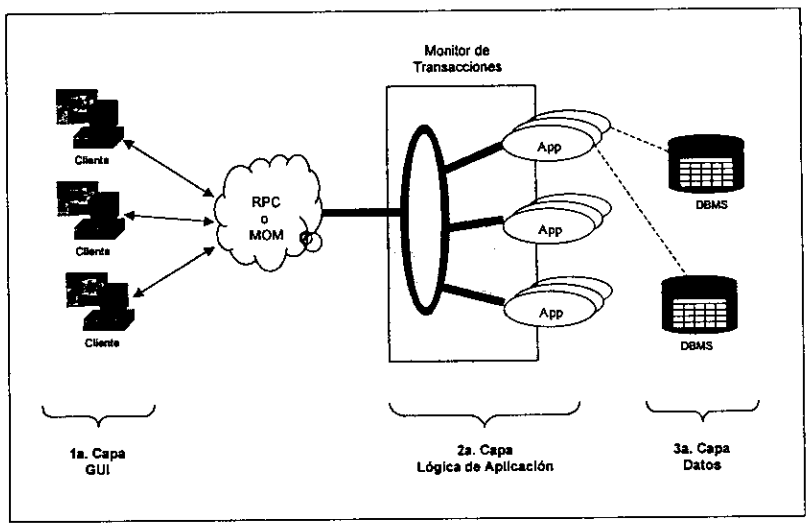
Los monitores de transacciones tienen un fundamento sólido. En los ambientes de mainframe, un monitor de transacciones es vendido junto con una base de datos de misión-crítica. Los ingenieros de los ambientes de mainframe se dieron cuenta que no se pueden crear aplicaciones críticas sin manejar o administrar los procesos que operan los datos. Los monitores de transacciones nacieron para manejar procesos y para administrar programas. Logran esto al descomponer las aplicaciones complejas en piezas de código denominadas servicios. Utilizando transacciones, un monitor de transacciones puede organizar varias piezas de software que no tienen conocimiento las unas de las otras y hacerlas trabajar al unísono. Esto es una función deseable en ambientes C/S. Estos ambientes eventualmente pueden manejar miles de trasacciones diariamente ejecutándose en cualquier parte de la arquitectura de la aplicación.

Así que dichos monitores de transacciones aparecieron primero en el ambiente de mainframe para proveer ambientes de ejecución robustos que pudieran soportar aplicaciones de procesos de transacciones en línea a gran escala – aplicaciones que requieren de respuesta inmediata y tienen una gran demanda de sus servicios. Desde entonces, OLTP se ha extendido a casi todos los tipos de aplicaciones de negocios – incluyedo hospitales, manufactura, sistemas de punto de venta, etc. Los monitores de transacciones proveen los servicios que mantienen estas aplicaciones OLTP, ejecutándose

en un ambiente de alta disponibilidad. Con OLTP moviendose a las arquitecturas C/S y plataformas abiertas, un nuevo conjunto de monitores de transacciones nació para permitir que las aplicaciones de misión-crítica fueran más nobles.

Podemos definir entonces a un Monitor de Transacciones como un mini sistema operativo para aplicaciones transaccionales. Es también un esqueleto para las aplicaciones tipo servidor de la capa intermedia, y hace las siguientes cosas muy bien:

- **Manejo de Procesos** que se encarga de activar los procesos servidores, distribución del trabajo a estos procesos, monitoreo de su ejecución, y balanceo de cargas de trabajo.
- **Manejo de Transacciones** que significa que el monitor garantiza las propiedades ACID a todos los programas que se ejecuten bajo su protección.
- **Manejo de comunicaciones C/S** que permite a los clientes (y servicios) invocar un componente de una 7aplicación en una variedad de formas – incluyendo petición-respuesta, conversaciones, encolamiento, publicación y suscripción, y notificación.



2.1 C/S de 3 Capas, Estilo con Monitor de Transacciones

Funciones de un Monitor de Transacciones

Los monitores de transacciones fueron introducidos para ejecutar la clase de aplicaciones que atienden a miles de clientes al día. Las aplicaciones logran esto debido a que el monitor provee de un ambiente en el cual el monitor se interpone entre las peticiones de los clientes y los recursos del servidor, de manera que puede manejar las transacciones, rutear las peticiones a través del sistema, balancear la carga de trabajo, y habilitar a la aplicación a recuperarse de determinadas fallas. Además de que mejoran el rendimiento general de la aplicación. Los monitores de transacciones manejan las transacciones desde el punto de inicio – típicamente el cliente – a través de uno o más servidores, y de regreso a quien originó la llamada. Y por supuesto se aseguran que la transacción sea llevada a cabo de manera adecuada.

El monitor maneja todo el tráfico que liga cientos (o miles) de clientes con los programas aplicativos y los recursos finales. Estos procesos ahora tienen una existencia por separado de la base de datos y del ambiente del cliente (GUI). Esto significa que se pueden distribuir en distintas máquinas y redes o donde tenga más sentido que existan.

Una de los beneficios de utilizar un monitor de transacciones, es que este se encarga de vigilar todos los aspectos del procedo distribuido, independientemente de los sistemas o los recursos de datos con los que se cuenta. Puede manejar los recursos en un solo servidor o múltiples servidores.

Los monitores de transacciones pueden ahorrar dinero. De acuerdo con Standish Group, esta tecnología ahorra alrededor del 30% en el costo total del sistema – dependiendo de la escala del sistema – cuando se tiene un enfoque hacia una base de datos centralizada. Adicionalmente, la investigación de Standish Group muestra que las compañías pueden ahorrar en los tiempos de desarrollo – hasta 40% o 50%. Los monitores, con el balanceo de cargas, también proveen mejor desempeño utilizando los mismos recursos; esto significa que se puede ejecutar la aplicación en hardware menos costoso. Finalmente, los monitores no atan la aplicación a soluciones específicas de bases de datos – lo cual hace el proceso de

adquisición más competitivo y ayuda al ahorro de los costos. Si se usa la interfase estándar de transacciones, se puede inclusive evitar el atarse a un monitor de transacciones específico.

Funcionamiento de una Aplicación con Monitor de Transacciones

Un monitor de transacciones provee el esqueleto o esquema que ayuda a construir, ejecutar y administrar una aplicación del tipo C/S, de manera que no se tiene que comenzar la construcción desde cero. Los monitores proveen una plataforma excelente para un desarrollo rápido, robusto y de alto desempeño.

Los monitores de transacciones proveen del esquema para el desarrollo de aplicaciones tipo C/S. Al mismo tiempo los vendedores de herramientas de desarrollo están soportando el protocolo de RPC's y haciendo el desarrollo con un monitor de transacciones transparente al programador. Por el lado del servidor, los monitores de transacciones, se enfocan a la creación de procedimientos modulares y por tanto reusables – llamados servicios – que encapsulen los recursos del manejador de datos. Un manejador de datos es una pieza de software que maneja recursos compartidos – como una base de datos, una cola persistente, o un sistema de archivos transaccional. Los monitores proveen de shells de servidor de propósito genérico (llamadas clases de servidores) que ejecutan los servicios de la capa intermedia.

El monitor de transacciones introduce un estilo de programación por eventos en el lado del servidor al permitir asociar los servicios – que actúan como manejadores de eventos – con los eventos de servidores. Se exporta la llamada a la función y no los datos mismos. Esto significa que se puede continuar añadiendo nuevas llamadas a funciones y permitir que el monitor sea el que distribuya dicha función entre múltiples servidores. Los monitores permiten además crear aplicaciones altamente complejas al poder simplemente añadir nuevos servicios.

Una característica de los monitores es que permiten que servicios que no tienen relación los unos con los otros puedan trabajar con propiedades ACID. ACID (Por su significado en inglés: Atomicity, Consistency, Isolation, Durability) son los principios bajo los cuales una aplicación transaccional se guía.

Propiedades ACID.

Desde el punto de vista del negocio, una transacción es una acción que cambia el estado de un objeto de negocio – por ejemplo, un cliente depositando dinero en su cuenta de cheques constituye una transacción bancaria. Técnicamente hablando, una transacción es una colección de acciones que están gobernadas por las propiedades ACID. Los monitores de transacciones y los ORB's proveen de estas propiedades a las aplicaciones. ACID – un termino acuñado por Andreas Reuter en 1983 – significa Atomicidad, Consistencia, Aislamiento, y Durabilidad.

La **Atomicidad** (Atomicity) significa que una transacción es una unidad de trabajo indivisible: todas las acciones se hacen, o no se hace ninguna de ellas. Es una operación donde se hace todo o no se hace nada. Las acciones que una transacción puede implicar, pueden ser, encolamiento de mensajes, actualizaciones a la base de datos, y el despliegue de resultados en la pantalla del cliente. La atomicidad esta definida desde la perspectiva del instigador de la transacción, generalmente el cliente.

La **Consistencia** (Consistency) significa que una vez que la transacción se ha ejecutado, debe dejar el sistema en un estado correcto o debe de abortar. Si la transacción no puede lograr dicho estado, debe regresar el sistema al punto inicial donde comenzó la transacción.

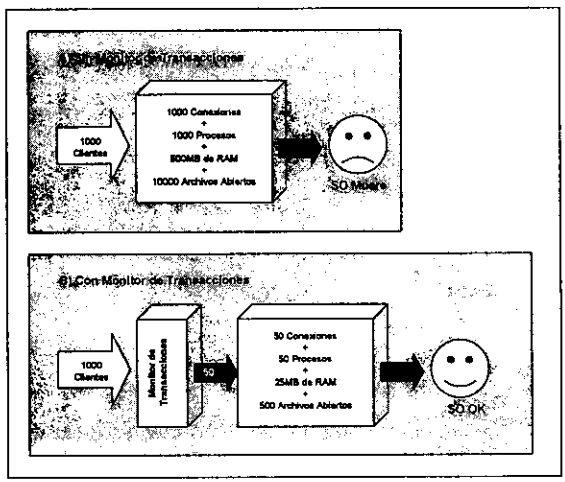
El **Aislamiento** (Isolation) significa que la ejecución de la transacción no sea afectada por la ejecución concurrente de otras transacciones. La transacción debe de serializar todo acceso a los recursos compartidos y garantizar que los programas concurrentes no corromperán las operaciones de cada uno. Un programa de múltiples usuarios ejecutándose bajo ambiente transaccional debe de comportarse de manera identica a un programa de un solo usuario. Los cambios que una transacción hace a un recurso compartido no deben de ser visibles fuera de la transacción hasta haber hecho el commit.

La **Durabilidad** (Durability) significa que los efectos de la transacción son permanentes después de que realiza el commit. Los cambios que una transacción haya hecho deben de sobrevivir a las fallas de los sistemas. El término persistente es un sinónimo de durable.

Los monitores permiten combinar los manejadores de recursos, de manera que se puede comenzar con un solo manejador de recursos y posteriormente, migrar a otro mientras se mantiene la inversión de la aplicación desarrollada. Todos los servicios utilizan la infraestructura de manejo de procesos del monitor de transacciones. En otras palabras, se pueden añadir recursos de servidores heterogéneos sin alterar la arquitectura de la aplicación.

Multiplexaje Lógico

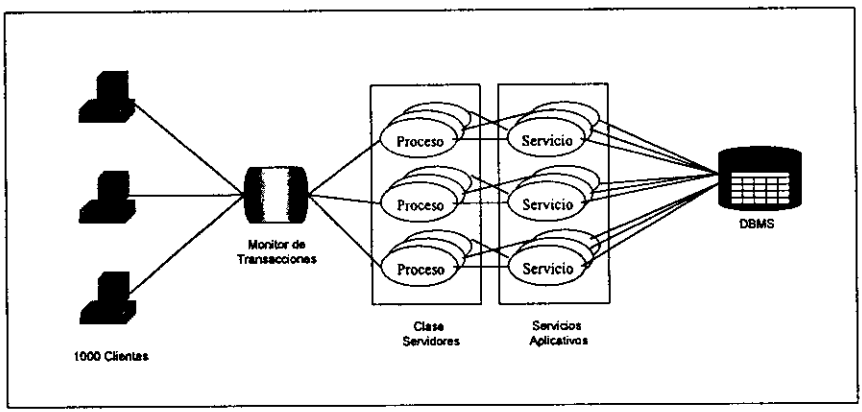
Como ya se ha mencionado, los monitores de transacciones fueron introducidos para ejecutar la clase de aplicaciones que pueden servir a cientos y hasta miles de clientes. Si cada uno de estos clientes tomara en un servidor todos los recursos que necesitara – típicamente una conexión de comunicación, medio megabyte de memoria, uno o dos procesos, y una docena de apuntadores a archivos – aún un mainframe tendría problemas al servir dichos recursos. Con un poco de suerte, no todos los clientes necesitarán el mismo servicio al mismo tiempo. Sin embargo, cuando eso sucede, requieren de una respuesta inmediata. Las personas por otro lado, tienen una tolerancia de espera de segundos, no más. Los monitores proveen del mini sistema operativo del que se habló anteriormente, y que conecta en tiempo real, estos miles de impacientes clientes a un conjunto de procesos compartidos. El monitor balancea entonces el uso de recursos entre los clientes bajo el esquema de demanda – esto es llamado multiplexaje lógico.



2.2 Porque un SO necesita un monitor de transacciones

El multiplexaje lógico es parte de lo que un monitor de transacciones debe de hacer para manejar de manera eficiente el lado servidor de una aplicación OLTP. El lado servidor de una aplicación OLTP es un conjunto de servicios que contiene un determinado número de funciones. El monitor asigna la ejecución de cada servicio a un servidor aplicativo, el cual es parte de un conjunto de procesos iniciados con anterioridad y que esperan peticiones para trabajar. El monitor balancea la carga de trabajo entre dichos procesos. Cada aplicación puede tener uno o más servidores aplicativos.

Cuando un cliente envía una petición, el monitor pasa dicha petición a un proceso disponible del conjunto de servidores aplicativos. El servidor aplicativo procesa dinámicamente la petición, invoca una determinada función según la petición, monitorea su ejecución, y regresa el resultado al cliente. Una vez completo este proceso, otro cliente puede reusar el proceso del servidor aplicativo. El sistema operativo mantiene en la memoria los servicios ya cargados, los cuales son compartidos por los procesos de los servidores aplicativos.



2.3 Multiplexaje Lógico

En esencia, el monitor elimina el requerimiento de un proceso por cliente al multiplexar de manera lógica las peticiones a los procesos de los servidores aplicativos. Si el número de las peticiones de los clientes excede el número de procesos en un servidor aplicativo, el monitor puede de manera dinámica iniciar nuevos procesos – lo cual es llamado balanceo de cargas. Algunos monitores sofisticados pueden distribuir el proceso entre varios CPU's en ambientes SMP o MPP. Parte del balanceo de carga de trabajo involucra el manejo de prioridades de las peticiones entrantes al sistema. El monitor hace dicho manejo al ejecutar servidores de alta prioridad y asignándolos de manera dinámica a los clientes.

Algo que es importante mencionar, las funciones que se ejecutan en poco tiempo, suelen organizarse en un servidor aplicativo de alta prioridad. Las funciones con trabajo tipo batch o de baja prioridad son asignadas en servidores aplicativos de baja prioridad. Se pueden particionar los servidores aplicativos por tipo, tiempo de respuesta, recursos que manejan, tolerancia de fallos, modos de interacción. Además de proveer el ya mencionado balance de cargas, el monitor de transacciones permiten el control manual de cuantos procesos estén disponibles.

Manejo de Transacciones

La disciplina transaccional fue introducida con los primeros monitores de transacciones para garantizar la robustez de aplicaciones con muchos usuarios. Estas aplicaciones debian de ser a prueba de fallos y altamente confiables. La unidad de manejo, ejecución, y recuperación es la transacción y los programas que invocan dicha transacción. El trabajo de un monitor de transacciones es garantizar que se cumplan las propiedades ACID, mientras se mantiene un nivel de proceso de transacciones alto; para lograr este último punto, debe de manejar la ejecución, distribución, y sincronización de la carga de las transacciones.

La coordinación de las transacciones es importante. Los clientes combinan y ajustan los componentes para implantar una transacción del negocio. Cada uno de estos componentes puede realizar una actualización a los datos. Por lo tanto se convierten en 'miniprogramas'. Como resultado, dichas actualizaciones necesitan estar coordinadas. Los monitores llevan a cabo esta tarea al poner a disposición de la aplicación las propiedades ACID. Por ejemplo, los componentes pueden actualizar la misma base de datos o distintas – quizás distintos tipos de bases de datos. ¿ Qué sucede si la transacción es detenida o falla durante el desarrollo de dichas actualizaciones ? El monitor asegura que todos las actualizaciones asociadas con una transacción abortada sean removidas o 'rolled back'. Puede inclusive tener este control aún cuando los componentes se encuentren en distintos servidores actualizando distintas bases de datos. Cuando los manejadores de recursos están comunicados en red, el monitor sincroniza todas la actualizaciones utilizando el protocolo de 'two-phase commit'.

Protocolo de Two-Phase Commit

Este protocolo sincroniza las actualizaciones de manera que, o todas fallan, o todas tienen éxito. Esto se logra al centralizar la decisión de llevar a cabo el commit, pero dando a cada participante el derecho del voto.

Cada implantación comercial de dicho protocolo tiene su propia variante – y no interoperan. Pero por supuesto, hay estándares que tratan de hacer trabajar el conjunto. En diciembre de 1992 – después de un ciclo de desarrollo de 5 años – ISO publicó su estándar OSI-TP que define rígidamente como se debe llevar a cabo el two-phase commit.

Así, OSI-TP permite que distintos monitores trabajen conjuntamente para sincronizar las transacciones. Los mecanismos de este protocolo son los siguientes:

En la Primera fase del commit, el nodo manejador del commit – también conocido como nodo raíz o coordinador de la transacción – envía el comando preparar-para-commit a todos los nodos subordinados que participan en la transacción. Los subordinados podrían haber extendido la transacción a otros nodos (o manejadores de recursos) a los cuales se debe de propagar el comando preparar-para-commit. Se convierte en un árbol de transacción, con el coordinador como raíz.

La primera fase del commit termina cuando el nodo raíz recibe las señales listo-para-commit de todos los subordinados directos que participan en la transacción. Esto significa que la transacción se ha llevado a cabo de manera exitosa en todos los nodos, y que están listos para

llevar a cabo el commit final. El nodo raíz registra el hecho en un log seguro, que se puede utilizar para recuperarse de una falla en el nodo raíz.

La segunda fase del commit inicia después que el nodo raíz hace la decisión de llevar a cabo el commit de la transacción – basado en el voto unánime del sí. Indica a los subordinados hacer el commit. Estos, a su vez, indican a sus correspondientes subordinados hacer lo mismo, y la orden barre el árbol.

La segunda fase del commit termina cuando todos los nodos involucrados han llevado a cabo el commit de su parte de la transacción de manera exitosa, y la han hecho durable. La raíz recibe todas las confirmaciones y puede indicar al cliente que la transacción ha sido completada.

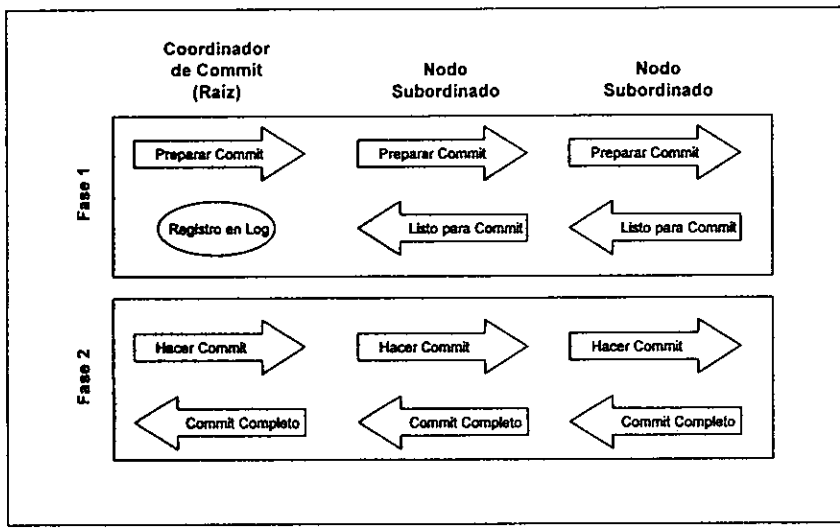
El two-phase commit aborta si uno de los participantes regresa una indicación de negación, significando esto, que su parte de la transacción ha fallado. En este caso, el nodo raíz indica a todos sus subordinados llevar a cabo un rollback. Y estos subordinados, indicaran lo mismo a sus subordinados.

La especificación XA de X/Open define un conjunto de API's que trabajan con el protocolo OSI-TP. Para participar en un protocolo de two-phase commit, el monitor y el manejador de recursos (como bases de datos y colas de mensajes) deben de mapear su protocolo privado de two-phase commit, con los comandos de XA. Deben de estar dispuestos además, a que alguien más pueda manejar la transacción – algo que no es usual. La especificación de XA permite que los participantes se retiren de la participación de la transacción global durante la fase 1 si no tienen que actualizar recursos. En XA, un monitor puede utilizar también un commit de una fase si esta tratando con un solo manejador de recursos.

A principios de 1996, la mayoría de los monitores podían fácilmente manejar transacciones que se extendían entre 100 máquinas de two-phase commit. Sin embargo, el protocolo de two-phase commit no es perfecto; estas son algunas de sus limitaciones:

Sobrecarga de trabajo es un factor que se añade a la aplicación por el intercambio de mensajes. El protocolo no tiene forma de discernir las transacciones de más valor que necesitan esta clase de protección, que las que pueden tolerar una falla y no necesitan protección. Genera mensajes para todas las transacciones, inclusive aquellas que sólo son de lectura. Así que los arquitectos mantienen aquellas operaciones de solo lectura, fuera de los límites de las transacciones globales.

Periodos de Falla, donde ciertas fallas pueden ser un problema. Por ejemplo, si el nodo raíz se pierde durante la primera fase del commit, los subordinados pueden quedar en desorden. ¿Quién se encarga de este problema? Siempre hay soluciones alternas, pero son trucos. Ayuda el hecho de utilizar algún software tolerante a fallas de hardware en el sistema, que auxilie al control de la transacción si un evento de este tipo se sucede.

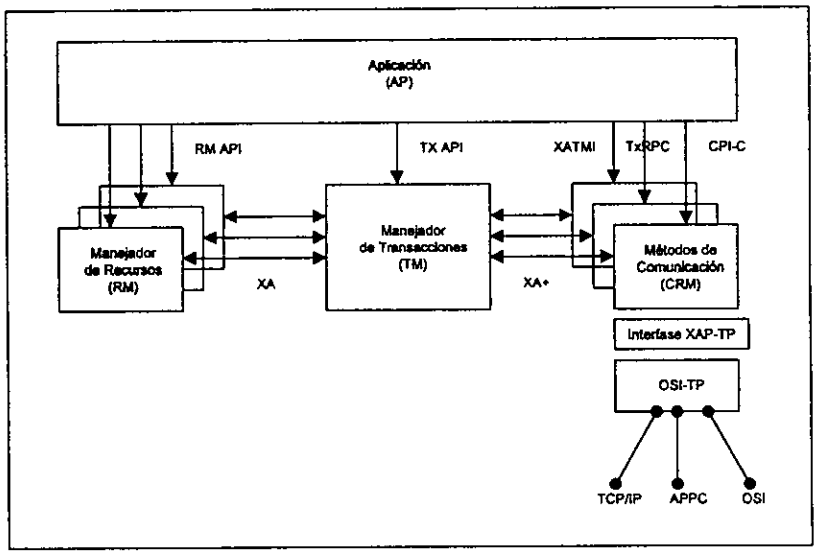


2.4 Mecanismo del protocolo Two-Phase Commit

Con los monitores, los programadores de la aplicación no necesitan preocuparse de cuestiones como la concurrencia, fallas, conexiones perdidas, balanceo de carga de trabajo, y la sincronización de los recursos a través de múltiples nodos. Todo esto es transparente al programador. En resumen y de manera sencilla, los monitores proveen las maquinarias de ejecución para la ejecución de transacciones del negocio – y lo llevan a cabo sobre el sistema operativo y hardware ordinario.

XA, OSI-TP, y Estándares de Transacciones

El conjunto de estándares de X/Open, ha definido un conjunto de especificaciones que permite a las aplicaciones, los manejadores de recursos (como bases de datos), y los manejadores de transacciones puedan sincronizar transacciones distribuidas. Este es el estándar llamado X/Open DTP (Por sus siglas en inglés: Distributed Transaction Processing). Ya que hay distintos componentes involucrados en una transacción, múltiples interfaces necesitan ser definidas. Estas son las más importantes de dichas interfaces:



2.5 Modelo de Referencia de Transacciones Distribuidas de X/Open 1994

RM API, que es utilizado por una aplicación para consultar y actualizar recursos que son poseídos por un manejador de recursos (Resource Manager - RM).

TX API, que es utilizado por una aplicación para señalizar al manejador de transacciones que está iniciando una transacción, que está lista para hacer commit, o que quiere abortar la transacción.

XA API, que es utilizado por el manejador de transacciones para coordinar actualizaciones entre manejadores de recursos. A través de esta interfase, el manejador de transacciones indica al manejador de recursos cuando preparar-para-commit, hacer commit, finalizar, o llevar a cabo un rollback de una transacción.

OSI-TP, que es un protocolo que permite que manejadores de transacciones heterogéneos trabajen en conjunto para coordinar transacciones.

XATMI, TxRPC, y CPI-C son interfaces de programación de comunicación transaccional.

Comunicación Transaccional.

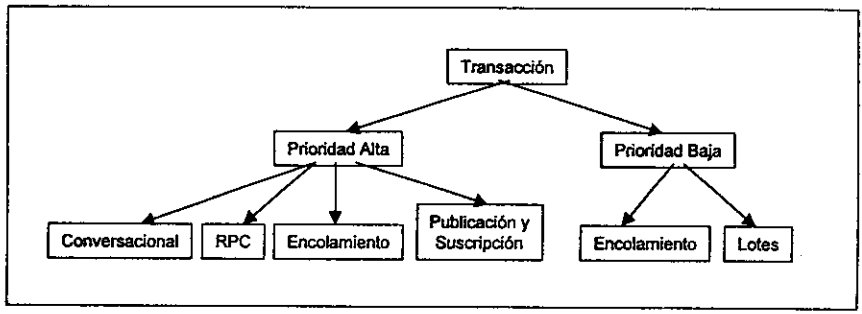
Los sistemas operativos deben de entender la naturaleza de los procesos y recursos que maneja. Esto aplica también con los monitores de transacciones – deben de proveer de un ambiente optimizado para la ejecución de las transacciones que corren bajo su control. Esto significa que debe de cargar los servidores, asignar de manera dinámica las peticiones de los clientes a los procesos correspondientes, recuperarse de fallas, regresar las respuestas a los clientes, y asegurarse que las transacciones del alta prioridad sean atendidas primero.

Uno de los trabajos de los monitores de transacciones es proveer de comunicación entre los clientes y los servidores – y entre los servidores mismos. Necesitan llevar esto a cabo mientras mantienen las propiedades ACID. ¿ Qué clase de comunicación transaccional provee un monitor de transacciones ? Típicamente soportan una de las siguientes: conversacional, RPC, encolamiento, publicación y suscripción, y proceso por lote (batch). Las transacciones en modo conversacional y RPC generalmente involucran un usuario humano que requiere de atención inmediata; se ejecutan en modo de alta prioridad. Las transacciones en modo de publicación y suscripción generalmente también se ejecutan con alta prioridad. Las transacciones en modo de lote generalmente se ejecutan con baja prioridad.

Los monitores de transacciones pareciera que utilizan estos mecanismos para su funcionamiento interno, en cierta manera lo hacen, sin embargo las versiones de las comunicaciones antes mencionadas de las que hacen uso los monitores, son versiones altamente sofisticadas. Muchos de los elementos que generan la alta complejidad de dichos mecanismos de comunicación son transparentes a los desarrolladores. Algunos de dichos elementos son los siguientes:

Delimitadores transaccionales. Estos permiten a un cliente especificar los limites del inicio y fin de una transacción. Los mecanismos actuales del commit son delegados a un servidor que pertenece al monitor

de transacciones – de otra manera, los clientes tendrían que manejar las propiedades ACID y registrar los eventos en los logs correspondientes, lo cual sería una tarea poco manejable y poco confiable.



2.6 Perfil de Transacciones en C/S

Intercambio de tres vías – bajo la mascara – entre el cliente, el servidor y el monitor (Manejador de transacciones). Una nueva transacción obtiene por medio de asignación un identificador único por el coordinador del monitor. Todos los intercambios de mensajes entre los participantes son etiquetados con el identificador de la transacción. El intercambio de mensajes permite al monitor tener registro de los recursos que participan en una transacción distribuida. Los monitores necesitan dicha información para coordinar el protocolo de two-phase commit con todos los participantes de la transacción.

Incluyen información del estado de la transacción en cada uno de los mensajes intercambiados. Esta información ayuda al monitor a identificar el estado de la transacción distribuida y decidir que es lo que sigue.

Proveen ruteo de servicios con base en las clases de servidores, carga de trabajo, recuperación de fallas y otros factores.

Los mecanismos avanzados han sido denominados como transaccionales. El factor que los distingue es que todos los manejadores de recursos y procesos invocados a través de estos mecanismos toman parte en la transacción. El monitor es informado de cualquier llamada a algún servicio; entonces utiliza dicha información para coordinar las acciones de todos los participantes, refuerza su conducta de acuerdo a las propiedades ACID, y los hace actuar como parte de la transacción. En contraste, los mecanismos tradicionales, pasan mensajes a procesos independientes que no están coordinados en transacción alguna.

Como ejemplos de estos mecanismos transaccionales al nivel comercial están los siguientes: En el campo de RPC's, Encina RPC Transaccional de Transarc/IBM, Tuxedo TxRPC de BEA, y CICS Interface de llamadas externas de IBM. En el campo de interfaces transaccionales en modo conversacional se encuentran en ATMI de Tuxedo, RSC de Tandem, APPC de IBM. Series MQ de IBM y MessageQ de BEA pertenecen al mercado de mercado transaccional de MOM. Algunos monitores también incluyen sus versiones propias de encolamiento: RQS de Encina, /Q en Tuxedo, colas de transito en CICS. Tuxedo tiene mecanimos de publicación y suscripción soportados por un componente llamado EventBroker. Finalmente, las implantaciones comerciales de los ORB´s, comienzan a aparecer en productos como Orbix de Iona, M3 de BEA y Component Broker de IBM.

Proceso de Transacciones Ligero vs Proceso de Transacciones Real (2-Capas vs 3-Capas).

Los stored procedures (procedimientos almacenados) son la competencia más directa que los monitores de transacciones enfrentan. Los vendedores de bases de datos introdujeron el concepto de stored procedures (SPs) para mejorar el rendimiento de la arquitectura de 2-capas. Los SPs consisten de sentencias SQL y lógica del negocio – típicamente implantados en el 4GL de la base de datos. El modelo de la base de datos no trata los procesos como entidades independientes – los procedimientos están almacenados y son ejecutados dentro de la base de datos.

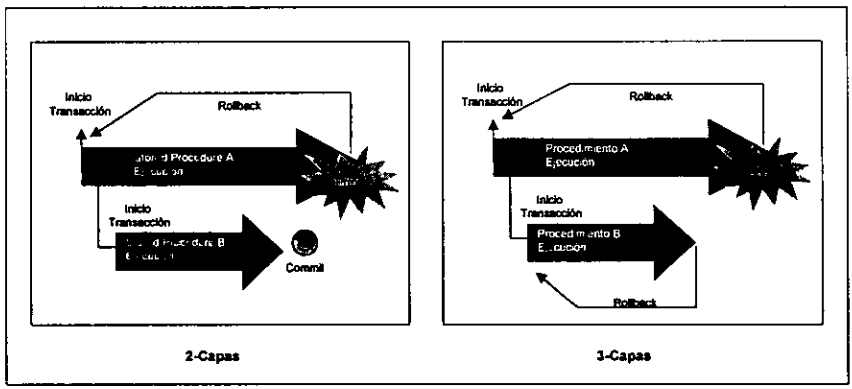
De alguna forma, esta variante integra algunas de las funciones de un monitor dentro de la base de datos. Sin embargo pocas funciones son las integradas y muchas las que no se incluyen. La inclusión de algunos elementos ayudó a reparar algunos problemas con las aplicaciones de 2-capas, pero también trajo muchos otros. En comparación con los monitores de transacciones, que ayudan a la construcción de aplicaciones de 3-capas, el proceso de transacciones ligero, tiene inconvenientes cuando se compara contra el proceso de transacciones real que un monitor de transacciones provee, aquí algunos puntos de comparación:

Alcance del Commit.

Un stored procedure esta escrito en un lenguaje propietario perteneciente a la base de datos – PL/SQL, Transarc, y otros – y esta almacenado dentro de la base de datos. Es una unidad transaccional, pero no puede participar en conjunto con otras unidades transaccionales en una transacción global. No puede invocar otra transacción y hacer que se ejecute dentro del contexto de una misma transacción. Por ejemplo, si un SP A muere después de haber invocado a un SP B, el trabajo llevado a cabo por el SP A

será manejado por la base de datos y se hará un rollback, mientras que el trabajo del SP B será exitoso, y se hará un commit de la operación. Esto es una violación a las propiedades ACID en su postulado de todo o nada. Esta limitación ocasiona escribir SPs largos que mantienen todo el trabajo dentro del alcance del commit. No ayuda a escribir módulos o reutilizar funciones.

Por el contrario, con el uso de un monitor de transacciones, los procedimientos escritos se desarrollan con lenguajes procedurales estándar. Fácilmente pueden proveer protección del tipo todo o nada. Por lo tanto, el manejo de transacciones globales en un monitor de transacciones en natural.

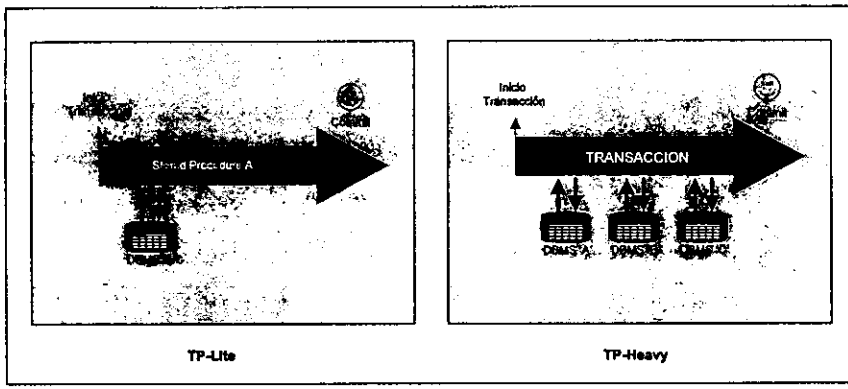


2.7 Alcance del Commit

Manejo de recursos heterogéneos.

Un SP puede controlar los recursos que se encuentran en la base de datos donde vive. No puede sincronizar o controlar recursos que estén en otra base de datos u otro manejador de recursos – ya sea local o remoto. Los monitores de transacciones pueden controlar varios recursos heterogéneos dentro del contexto de una transacción global.

Algunas bases de datos, pueden aplicar el protocolo two-phase commit, a múltiples bases de datos, generalmente las propias. Open Gateway de Oracle es un producto que permite el manejo del protocolo two-phase aún entre bases de datos heterogéneas compatibles con el estándar de XA. Sin embargo los gateways asumen que un SP dentro de la base de datos representa la aplicación (y también el punto de origen de una transacción). Los gateways no permiten que múltiples aplicaciones (o SPs) participen en una transacción. Una desventaja más es que los gateways amarran las posibilidades del ambiente a los SPs de la base de datos utilizada. Con un monitor de transacciones su logra la independencia de dichos procedimientos.

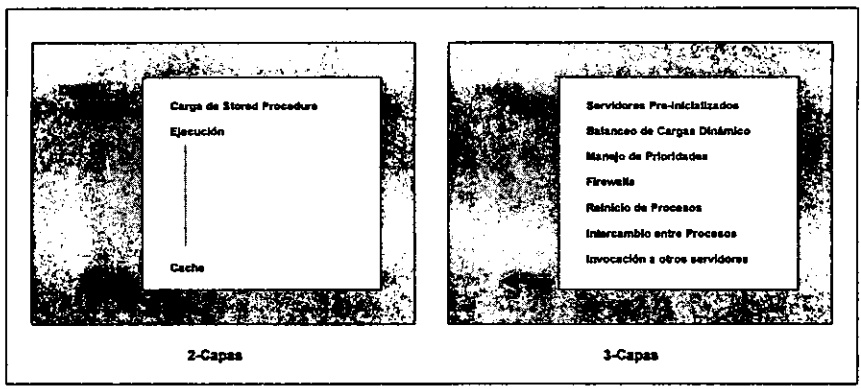


2.8 Sincronizando Manejadores de Recursos Heterogéneos

Manejo de Procesos.

Un SP es invocado, ejecutado bajo el control de la protección de las propiedades ACID (En commit de una fase), y podría ser capturado en memoria para uso futuro. En contraste, los procesos sometidos a un monitor de transacciones, son iniciados como clases de servidor. Dichos procesos representan la lógica

del negocio – por lo cual están listos a actuar una vez que la petición de un cliente es recibida. La carga es balanceada entre dichos procesos. Si la carga de trabajo rebasa la capacidad de trabajo ofrecida por los procesos, nuevos procesos son iniciados. Las clases de servidor soportan prioridades y otros atributos inherentes a un servicio. Dichos procesos están protegidos los unos de los otros para evitar interferencias. Si un proceso muere, es reiniciado o la transacción es dirigida a otro proceso. El monitor constantemente supervisa el ambiente.



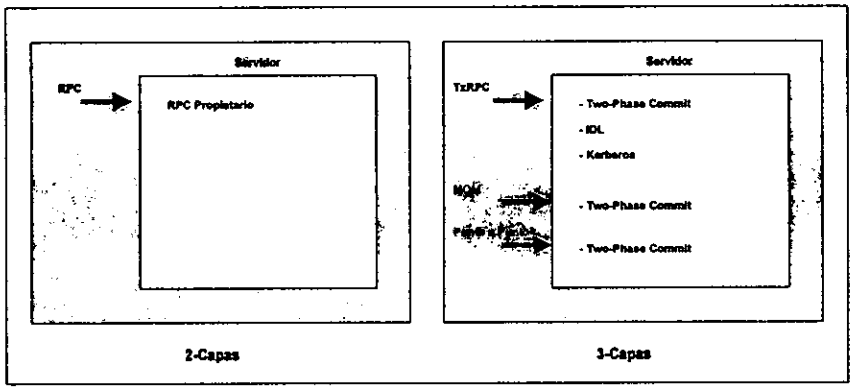
2.9 Manejo de Procesos, 2-Capas vs 3-Capas

Invocaciones C/S.

La invocación de un SP es algo que no se apega a un estándar. Cada base de datos provee de sus mecanismos para poder invocar los SPs que existen dentro de la base de datos. Dichos mecanismos de invocación no están definidos usando un lenguaje de definición de interfase (IDL – Interface Definition Language). No están integrados con la seguridad, y los servicios de autenticación. Estas omisiones ocasiona que los SP no sean una alternativa viable para transacciones globales. Los enlaces de

comunicación no son reiniciados de manera automática, no hay balanceo de carga de trabajo, y no hay protección transaccional. Un último punto, no soportan diferentes mecanismos de comunicación.

En un ambiente con monitor de transacciones, las opciones de comunicación son varias. Las sesiones de los clientes son autenticadas. Los clientes accesan los componentes a través de nombres lógicos. Los clientes pueden definir los limites de la transacción, pueden invocar múltiples componentes, e incluir sistemas heredados dentro de una transacción.



2.10 Invocación en Cliente/Servidor

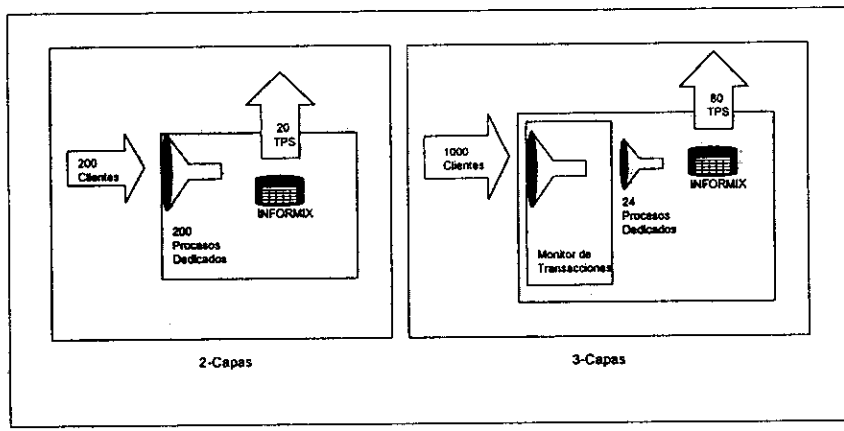
Rendimiento.

Debido a que reducen el tráfico de la red, los SPs son más rápidos. Sin embargo, no tienen un rendimiento tan bueno como los procesos manejados por un monitor de transacciones, especialmente con bastante carga de trabajo. Esto es debido a que los SPs son interpretados en la base de datos, y la mayoría de los lenguajes interpretados son lentos. En 1997, una prueba de rendimiento con bases de

datos, con transacciones TPC-C, alcanzó sus mejores números en rendimiento cuando se utilizó un monitor de transacciones.

ALR	Revolution 6X6	MS SQL Server	Tuxedo	10,665.53
Bull	ESCALA P4404-HE	Oracle	Tuxedo	14,285.97
Compaq	ProLiant 7000 6/200	MS SQL Server	Tuxedo	11,055.70
Digital	AlphaServer 8400 5/350	Oracle	Tuxedo	30,390.00
Digital	AlphaServer 8400 5/350	Sybase	Tuxedo	14,176.00
HP	9000 U2200	Sybase 11	Tuxedo	39,469.47
HP	9000 K570	Sybase 11	Tuxedo	21,358.00
Dell	Power Edge	MS SQL Server	Tuxedo	10,984.07
IBM	RS 6000 Cluster 4 Server	Oracle	Tuxedo	14,285.87
Motorola	Motorola Series MP MP601	Oracle	Tuxedo	3,512.97
SGI	Origin2000 Server	Informix	Tuxedo	25,309.20
SNI	RM600 Model620	Informix	Tuxedo	6,269.67
Sun	Ultra Enterprise 6000	Oracle 8.0.3	Tuxedo	31,147.04
Sun	Ultra Enterprise 6000	Oracle	Tuxedo	23,143.65
Sun	Enterprise 6000	Oracle 8.0.3	Tuxedo	51,871.62
Tandem	Integrity NR/4436 Server	Informix	Tuxedo	6,343.78
Unisys	Aquanta HS/6 Server	MS SQL Server	Tuxedo	13,089.30

El uso de un monitor de transacciones también impacta en los gastos, ya que permite ahorrar al proveer un sistema más eficiente. En esencia, el monitor de transacciones libera a la base de datos al multiplexar las peticiones de los clientes. Actúa como un multiplexor lógico montado sobre la base de datos. Adicionalmente, los procesos ejecutados por el monitor, son procesos compilados, por lo que se ejecutan con mayor rapidez.



2.11 Rendimiento, 2-Capas vs 3-Capas

Capitulo 3

En este capitulo se presentan los distintos paradigmas de comunicación, con los cuales puede ser implantada una aplicación distribuida.

Comunicación y Paradigmas de Administración para Aplicaciones Distribuidas.

Este capitulo habla acerca de las varias técnicas utilizadas por los componentes de las aplicaciones distribuidas para cooperar. Los componentes de una aplicación no pueden cooperar a menos que se comuniquen – la comunicación ocurre por medio de los mensajes que envían. El intercambio de mensajes puede ser llevado a cabo por varias razones en varias formas.

Así mismo, comienza por definir algunos términos y procede con las técnicas de comunicación más comunes. También describe técnicas para estructurar los datos que son transmitidos en módulos aplicativos que residen en computadoras con representación binaria distinta. El manejo de errores es una de las partes más difíciles al programar aplicaciones distribuidas, en este capitulo también se describen técnicas como time-out y transacciones, que facilitan la tarea de los programadores cuando las condiciones de error se presentan.

Posteriormente se tratan los términos y técnicas para la construcción de software administrativo para aplicaciones distribuidas. Finalmente se introducen algunos términos relacionados con la seguridad, que es relevante para las aplicaciones distribuidas.

Comunicación en una Aplicación.

La cooperación efectiva entre los módulos de software, requiere comunicación precisa entre ellos. Se introducen aquí conceptos utilizados en la comunicación entre módulos y se describen los principios del diseño de interfase utilizados para prevenir un mal funcionamiento en la comunicación.

Terminología.

Un software consiste de código, y el código esta conformado por una secuencia de sentencias que manipulan variables, constructores lógicos (pruebas condicionales, ciclos, etc.), y que invocan a piezas de software prefabicadas; estas últimas piezas son denominadas subrutinas, rutinas, funciones, primitivas, o procedimientos. Algunas ocasiones, un conjunto de procedimientos es utilizado para cumplir una tarea, como por ejemplo, proveer acceso a un archivo del sistema. Un API (Application Programming Interface) es la definición del formato de llamadas a un conjunto de procedimientos que realizan determinadas tareas. Un API, especifica la entrada, salida y reglas que norman las llamadas a los procedimientos. Por ejemplo, el acceso a un archivo generalmente involucra procedimientos para abrir y cerrar archivos, leer datos de estos, escribir datos a estos. Estos procedimientos conforman un API.

Un API de comunicaciones es una interfase a un conjunto de procedimientos que pueden ser utilizados para comunicar a varios módulos de software entre si. El software que implanta el API de comunicaciones es generalmente llamado 'pila de comunicaciones'. Este término deriva del modelo de comunicaciones OSI (Open Systems Interconnect) donde los módulos de software se comunican al invocar capas de funciones 'apiladas'. Los módulos se comunican enviándose mensajes. Los formatos y la secuencia de dichos mensajes que pueden intercambiados por un conjunto de módulos son llamados 'protocolos de comunicación' (o sólo protocolos). Así, en una computadora que inicia una comunicación,

un API de comunicaciones provee el software que inicia la comunicación con una interfase a una pila de comunicaciones, y la pila de comunicaciones emite un protocolo.

Para que una comunicación tenga sentido, el protocolo debe de ser aceptado por el nodo receptor. La pila en el nodo receptor acepta los mensajes de protocolo y los presenta a la aplicación a través de un API de comunicación en dicho sistema. Usualmente, el software que reside en ambos lados, utilizan el mismo API de comunicaciones. Deben de utilizar el mismo protocolo; de otra manera, un lado no entiende lo que habla el otro lado. El protocolo de la aplicación contiene los mensajes que las aplicaciones desean intercambiar. El protocolo de red es la representación de los datos de la aplicación en la red. El protocolo de red contendrá toda la información de la aplicación, pero podrá contener información adicional, por ejemplo, chequeos para asegurar que los datos son transmitidos de manera correcta.

La comunicación entre componentes o módulos de software en computadoras distintas presentan un problema: los datos pueden no tener la misma representación digital. Para que la comunicación sea efectiva, los datos deben de ser transformados de su representación original a la representación equivalente en donde serán procesados. Esta conversión es generalmente conocida como el 'servicio de presentación' (presentation service), y no debe de ser confundido con el software que genera imágenes en terminales.

Un 'paradigma' es un estilo, o forma, de hacer algo. Los paradigmas de comunicación son estilos de comunicación. Para entender dichos paradigmas, basta mirar la forma en que se comunican las personas. Las personas se comunican en forma declarativa ("X ha pasado"), imperativa ("Haz X"), y por petición ("¿ Me puedes decir X?"). La comunicación es algunas veces directa, por ejemplo, en una conversación telefónica; algunas veces es indirecta e iniciada unilateralmente por alguna de las partes, por ejemplo, una máquina contestadora. Algunas se llevan a cabo entre dos individuos (uno a uno), algunas veces entre un grupo de miembros (muchos a muchos), y algunas veces es un anuncio de una

persona a muchas. Algunas ocasiones la comunicación es corta y discreta, por ejemplo, cuando la compañía de luz envía el recibo de pago y el cliente paga. Puede ser continua u orientada al diálogo. En ocasiones utiliza pilas de comunicación básicas, por ejemplo, la voz humana utiliza el aire, y algunas veces es transmitida a través de comunicaciones con tecnología sofisticada, por ejemplo, enlaces de satelite.

Los sistemas distribuidos conducen los negocios de los humanos, así que es natural en su implantación el imitar los paradigmas de comunicación de las personas.

API de Comunicaciones.

El API de comunicaciones es una herramienta fundamental utilizada por los programadores para expresar la interacción de los módulos que conforman la aplicación distribuida. Un buen API de comunicaciones debe tener las siguientes características:

Debe de ser apropiado para la tarea de comunicaciones que pretende cumplir. Significando esto que debe de manejar las funciones de los tipos de comunicación más necesitadas en una aplicación distribuida. La interfase debe de crecer en complejidad si las tareas de comunicación crecen en complejidad. Las cosas simples deben de ser sencillas de hacer, y las más complejas requerirán de una interfase más compleja.

Debe de ser consistente en la forma de aceptar los argumentos, regresar datos, y en el manejo de errores. La facilidad de manejo de errores, en particular, requiere de un diseño cuidadoso; ya que dicho manejo se encuentra entre lo más difícil que un programador enfrenta.

Debe de ser intuitivo. Significa esto, que el programador no debe de ser sorprendido con el resultado de una comunicación exitosa o fallida, y el uso del API no deberá ayudar a generar errores.

Ya se ha mencionado un área donde un API de comunicaciones ayuda al programador: los servicios de presentación interconstruidos resuelven el problema de la presentación de datos entre computadoras heterogéneas. Otra área donde un API de comunicaciones simplifica la tarea de programación es la 'transparencia de locación'. Básicamente, las partes que conforman una aplicación se hablan entre sí, independientemente de donde se localizan dichas partes. Los servicios de presentación y la transparencia de locación son indicativos de un API de comunicaciones de alto nivel. De hecho, entre más alto el nivel de dicho API, la aplicación se encuentra en un ambiente donde se siente menos distribuida. También, los diseñadores y programadores de la aplicación se pueden concentrar más en las reglas de negocio y despreocuparse de los detalles de comunicación a bajo nivel.

Petición / Respuesta.

Una de las más simples interacciones entre programas de computadoras es el paradigma de la petición/respuesta. En este estilo de programación, un programa, llamado el cliente, pide a otro programa, el servidor, hacer algo. La petición puede ser el proveer alguna información o realizar alguna acción y reportar sobre el éxito o falla de esta. El servidor realiza la acción y generalmente regresa un resultado. El servidor es quien cumple un servicio en nombre del cliente. En un sistema bien diseñado, el cliente que hace una petición a un servicio, debe de saber:

- Qué datos requiere el servicio
- Que datos regresará el servicio.

El cliente no debe de saber:

- Donde se localiza el servicio.
- Como procesa el servicio la petición.
- Como fue creado el servicio.

Los primeros dos principios forman el contrato entre el cliente y el servidor. El tercer principio, llamado transparencia de locación, permite al servicio el ser movido a una nueva locación sin que el cliente sepa. Por ejemplo, en la computadora en donde se esta ejecutando el servicio, dicho servicio necesita ser desactivado por cuestiones de mantenimiento. El cuarto principio es llamado 'transparencia en la implantación' y exhibe 'encapsulación de información'. Permite la re-implantación del servicio sin cambios al cliente, independientemente si un algoritmo mejor para el servicio es desarrollado.

La implantación de un servicio puede ser tal que este invoque a otros servicios para llevar a cabo su trabajo. Cuando esto sucede, el servidor se convierte en un cliente, esto es, la noción de cliente y servidor son roles que los módulos de software asumen en la interacción que se lleva a cabo entre ellos. Por ejemplo, un servicio de Transferencia de fondos, utiliza otros servicios (Depósito y Débito) para llevar a cabo su trabajo.

Llamadas a Procedimientos.

Un número de métodos pueden ser utilizados para crear una conducta de petición/respuesta. En una llamada simple a un procedimiento, una invocación provee de argumentos a un procedimiento invocado. El control se transfiere al procedimiento invocado, y el que invocó suspende su operación hasta que el procedimiento invocado regresa el control de la petición, es decir hasta que esta completo. El procedimiento invocado realiza su función y regresa un resultado. La ejecución regresa al procedimiento que invocó, una sentencia después de la llamada que invocó al procedimiento. Aquí, ambos, el procedimiento que invoca y el procedimiento invocado, son ejecutados en la misma dirección de espacio, la invocación es síncrona, y la comunicación entre los procedimientos es arreglada por el sistema de compilación. El sistema localiza el procedimiento llamado en tiempo de ejecución y crea las áreas de paso y regreso de parámetros dentro de una dirección de espacio común para la entrada y salida del procedimiento invocado.

Un programador puede crear interacciones petición / respuesta al pasar mensajes entre módulos que se ejecutan en diferentes direcciones de espacio, posiblemente en diferentes computadoras. Esto se logra gracias al uso de un API de comunicaciones, o puede ser realizado por un sistema de compilación que convierte las llamadas a procedimientos de una dirección de espacio a llamadas de procedimientos en otra dirección de espacio. Este último caso es llamado 'Llamado de Procedimientos Remotos' (RPC – Remote Procedure Call).

Elementos de una comunicación Petición / Respuesta.

Los elementos importantes de este tipo de comunicación incluyen los siguientes:

- Un API de petición / respuesta que provee interfases para requerir que un servicio sea ejecutado, permite que los datos sean pasados a un servicio, y provee de datos de resultado que provienen de la ejecución del servicio.
- El módulo que invoca (llamado cliente) envía la petición a un servidor.
- El módulo invocado (llamado servidor) procesa la petición recibida de parte del cliente.
- El protocolo de comunicación es tan simple como ' una petición / respuesta'.
- Los datos (si hay tales) de una petición son transferidos del cliente al servidor.
- Los datos (si hay tales) de respuesta son regresados del servidor al cliente.
- La comunicación (en su forma básica) es síncrona al cliente – cuando un cliente hace una petición, no continuará su ejecución hasta que una respuesta del servidor haya sido recibida, esto es, el proceso del cliente esta sincronizado con el proceso del servidor.
- El método por el cual las condiciones de error son señalizadas a los módulos de comunicación, o es el regreso de errores en la llamada o son excepciones presentadas por el sistema de ejecución.

Este último punto es importante, ya que una simple interacción como una petición / respuesta está propensa a posibilidades de error, como las siguientes:

- ¿ Qué sucede si el servidor no existe cuando la petición es hecha ?
- ¿ Qué sucede si el cliente no existe cuando el servidor envía la respuesta ?
- ¿ Qué sucede si la petición contiene datos que el servido no entiende ?
- ¿ Qué sucede si la respuesta contiene datos que el cliente no entiende ?

- ¿ Qué sucede si el servidor termina de manera anormal mientras la petición se procesaba ?, ¿ Sabrá el cliente de dicha situación ?, ¿ Esperará indefinidamente por un resultado ?
- ¿ Qué sucede si la línea de comunicación es interrumpida mientras se transmite información ?

En estos casos, se requiere que el programador de la aplicación sea notificado de la condición de error. También se necesita que el sistema 'limpie' cualquier desorden que haya sido creado como resultado de una condición de error. Esto último se logra con una técnica denominada transacción.

Petición / Respuesta Asíncrona.

Algunas ocasiones un cliente puede llevar a cabo un proceso mientras el servidor esta ejecutando una petición. Por ejemplo en una aplicación bancaria que permite la transferencia de dinero de una cuenta de ahorros a una cuenta de cheques. La transferencia consta de dos acciones: obtener dinero de una cuenta y cargar dicho dinero en otra cuenta. Sin embargo no hay razón para que dichas acciones no puedan realizarse en paralelo, mientras ambas operaciones puedan ser supervisadas y haya un método para asegurarse que ambas o ninguna operación sucedan.

Este tipo de proceso permite que un módulo distribuya las peticiones en paralelo a un conjunto de módulos servidores. Dicha distribución reduce los tiempos de respuesta al cliente. Un API de comunicaciones que permite operaciones en paralelo también requiere funciones para determinar cuando la respuesta de dichas peticiones están disponibles. Típicamente la sintaxis para poder hacer una llamada se apega al tipo síncrono.. Esto es, que el proceso que generó dicha llamada no recupera el control hasta haber recibido una respuesta. Una forma de lograr la distribución de las peticiones en forma paralela es creando contextos de ejecución, llamados hilos de ejecución o threads, en el módulo que inicia las llamadas. Este thread separado puede entonces realizar llamadas síncronas por su cuenta, mientras que el contexto que lo creo puede hacer sus propias llamadas síncronas.

El proceso en paralelo de peticiones, ayuda a que los clientes puedan llevar a cabo otro proceso, mientras la petición que se envío es procesada en el servidor, por ejemplo, el proceso de las subsiguientes peticiones. Posteriormente el cliente se sincroniza con la respuesta a la petición al checar o esperar por una respuesta. Es importante señalar que la forma asíncrona no es visible al servidor. El servidor recibe una petición, la procesa, y regresa un resultado. No sabe si el cliente esta bloqueado esperando por una respuesta o si ha enviado otras peticiones.

Proceso en Etapas.

Al esconder la operación de un servidor al cliente, otra forma de proceso se puede generar en el sistema de comunicación. Un servicio que es invocado por un cliente puede procesar parte de la petición y delegar el resto del proceso a otro servidor. El delegado completa el proceso y responde al cliente. Mientras el delegado está trabajando en el proceso restante de la petición, el servidor original puede comenzar a trabajar en la primera parte del proceso de otra petición de otro cliente. Cada etapa del proceso hace alguna parte del trabajo de la operación total y pasa el resultado a la siguiente etapa. La etapa final responde al cliente.

Este tipo de proceso puede ser utilizado por ejemplo, por un sistema de tarjeta de crédito que necesita autorizar el uso de una tarjeta. Más aún, supongamos que el cliente necesita proveer de un identificador válido. La autorización consistiría de dos pasos:

- Asegurar que el cliente ha provisto el identificador adecuado.
- Asegurar que el cliente tiene crédito para la operación.

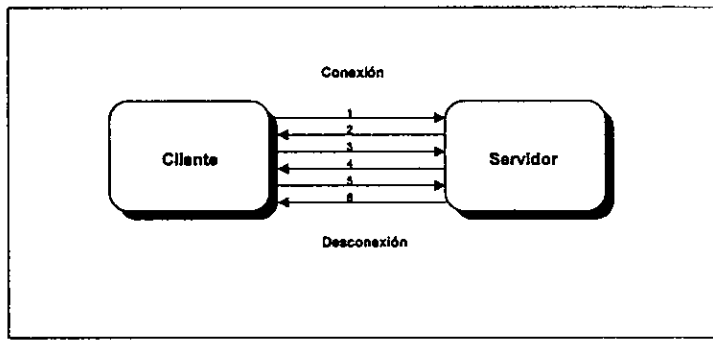
Estas reglas de autorización pueden ser ejecutadas en dos etapas. El cliente acepta los datos de la tarjeta de crédito que esta procesando. El cliente llama a un servicio de 'AUTORIZACION', y espera por una respuesta, NO o SI. El servidor de AUTORIZACION lleva acabo las siguientes funciones:

- Busca en la base de datos por el identificador del cliente. Si no se encuentra dicho identificado o esta incorrecto la respuesta es NO hacia el cliente. Si el cliente ha provisto el identificador de manera correcta, el servidor de AUTORIZACION invoca al servicio de CONTROL_DE_CREDITO.
- El servicio de CONTROL_DE_CREDITO checa el límite de crédito del cliente. Mientras esta operación sucede, el servicio de AUTORIZACION atiende la petición de otro cliente.
- El servicio de CONTROL_DE_CREDITO responde al cliente. Si el monto del crédito no es excedido ,la operación se autoriza; si el monto es sobrepasado, la operación se rechaza.

Este tipo de proceso es transparente al cliente, y permite mejores tiempos de respuesta al cliente, así como mejor capacidad de proceso de las peticiones que llegan al sistema.

Conversaciones.

Cuando se ordena una comida en un restaurante, el cliente y el camarero establecen un dialogo. Múltiples interacciones se harán sobre varios elementos, y dependiendo de las selecciones particulares, otras cuestiones y respuestas se desarrollaran. Finalmente, ambos elementos partícipes de la conversación llegan a un entendimiento sobre la orden de la comida. De la misma manera, en algunas ocasiones se da el caso que un número de mensajes necesitan ser intercambiados entre módulos de software para completar una operación del negocio. En tal situación, se dice que los módulos establecen una conversación. La noción de estado esta implícita en una conversación. Cuando uno de los elementos se comunica con su contraparte, ambos elementos recuerdan el punto (es decir, el estado) al cual la conversación ha progresado, e interacciones subsecuentes pueden suceder a partir de dicho punto. En la situación del restaurante mencionada anteriormente, el cliente y el camarero están conscientes del progreso de la conversación (es decir, de las preguntas y respuestas) referentes al menú. En el caso del software, el estado esta representado por los valores asociados con los datos (por ejemplo, variables) y las locaciones de los contadores de instrucciones en los módulos conversacionales.



3.1 Paradigma de Programación Conversacional

El flujo de datos entre los módulos que establecen una conversación tiene la siguiente lógica:

- El cliente inicia la comunicación, vía un mensaje de conexión enviado al servidor (flujo 1).
- En respuesta, el servidor envía dos mensajes al cliente (flujo 2 y flujo 3).
- El cliente envía otro mensaje al servidor (flujo 4), el cual envía un solo mensaje de respuesta (flujo 5).
- La secuencia es terminada por el servidor enviando un mensaje de desconexión al cliente (flujo 6).

Un uso común de las conversaciones es segmentar el contenido de un mensaje del servidor al cliente. En esta situación, un cliente requiere información de un servidor. El servidor tiene una gran cantidad de datos por enviar, tantos, que el cliente no puede recibir todos a la vez. En este caso, el servidor puede enviar parte de los datos e indicar al cliente que tiene todavía más datos por enviar. El cliente acepta el primer paquete de datos, los procesa, y cuando está listo, requiere al servidor para que le sea enviado otro tanto de los datos pendientes de recibir. Aquí, el servidor mantiene varios elementos como parte del estado, incluyendo el hecho de que hay mas datos por enviar y el punto de inicio de los datos que no han sido enviados. Cuando el cliente requiere más datos, el servidor comienza el envío desde este punto.

Si el protocolo entre el cliente y el servidor permite al cliente requerir información enviada con anterioridad, el servidor también tendrá que guardar dichos datos previamente enviados, junto con algunas variables de estado que indiquen que y donde se encuentran dichos datos.

Protocolos de aplicación para conversaciones.

El protocolo para la petición/respuesta es muy explícito: El cliente pide, el servidor responde. En un dialogo, sin embargo, múltiples mensajes pueden estar fluyendo en ambas direcciones entre los módulos que se están comunicando. Para una conducta correcta, algunas reglas deben de ser observadas. En el ejemplo de la segmentación del contenido del mensaje mencionado anteriormente, no tendría propósito el hecho de que el servidor sólo enviara mensajes al cliente. El propósito de la segmentación radica en que

el servidor contenga la información hasta que el cliente este listo para aceptar dicha información. El conjunto de instrucciones por las cuales una aplicación conversacional intercambia mensajes es llamado un *protocolo de aplicación*. Además de especificar el formato de los mensajes, los protocolos de aplicación indican los puntos permisibles para que alguno de los elementos participantes envíe datos. Por ejemplo, es una violación de protocolo para ambos lados de la conversación el estar esperando por un mensaje de su contraparte.

Terminación de la conversación.

En la interacción petición/respuesta, la interacción termina normalmente por el cliente cuando recibe respuesta del servidor. Para el servidor, ésta termina cuando la respuesta ha sido enviada al cliente. La terminación anormal puede ocurrir cuando, por ejemplo, el servidor falla en el proceso de la petición del cliente sin regresar una respuesta. En tal caso, el cliente:

- Espera el resultado por siempre, o
- Sabe de la falla del servidor por algún mecanismo, o
- Se cansa de esperar, o
- Algún otro software le indica que pare de esperar por la respuesta.

Para una conversación, la terminación se sucede normalmente por un acuerdo entre los módulos participantes. Un lado u otro terminará la conversación, que es presumiblemente aceptado por la contraparte. Si algún elemento no estará disponible más, es necesario notificar a la contraparte, para que cese el envío de mensajes. Un caso especial sucede cuando el canal de comunicación entre los módulos que conversaban se vuelve inoperante. Aquí, cada elemento ve desaparecer a su contraparte, y ambos necesitan ser notificados que la conversación no puede continuar.

El Contexto.

Cuando dos módulos de software están conversando, es necesario que mantengan un estado. De otra manera, sufren de amnesia, en cuyo caso, cada mensaje necesita contener un historial de la conversación hasta ese punto. Tal situación es difícil de programar. El estado está representado por el cambio en los datos de los módulos. Así, en el ejemplo previo, la segmentación del contenido del mensaje, el servidor recuerda la cantidad de datos que ha enviado al cliente. Una forma de control de estado es la locación del contador de instrucciones en un módulo conversacional. El contador de instrucciones indica que instrucción se ejecutará a continuación, así se deduce lo que el módulo espera que suceda a continuación.

Los elementos de una comunicación conversacional incluyen los siguientes:

Un API conversacional, que es utilizado por las aplicaciones para iniciar, intercambiar mensajes y terminar una conversación.
El módulo que inicia la conversación envía una petición de inicio a un módulo que será contraparte en la conversación. Esta petición puede contener datos específicos de la aplicación.
El protocolo de comunicación esta determinado por la aplicación. En las conversaciones alternas de dos vías, cada elemento participante es capaz de enviar mensajes, pero en un momento dado, sólo a uno le es permitido enviar un mensaje.
La comunicación es asíncrona para el que envía. Esto significa que cuando el elemento que controla la conversación envía un mensaje, puede continuar su ejecución, antes de que su contraparte reciba dicho mensaje. De hecho, puede enviar múltiples mensajes a su contraparte antes de ceder el privilegio de enviar mensajes a este último. Quien envía un mensaje, no sabe cuando este ha sido recibido. La sincronización debe de ser lograda por el protocolo de aplicación. Por ejemplo, el que envía, cede el control para enviar mensajes a su contraparte, y esperar un acuse por parte de este, de que ha recibido y procesado el mensaje previo.

La conversación normalmente es terminada por acuerdo. Cada lado puede terminar de manera anormal al desconectarse. Una conversación normal, termina cuando el servicio conversacional regresa de la ejecución con un indicador de éxito. Una terminación anormal se presenta cuando el cliente cuando este invoca una función de desconexión. El servicio conversacional termina de manera anormal al regresar con indicador de falla.

El método por el cual las condiciones de error son señalizadas a los módulos es por regreso de errores en la llamada o como manejo de excepciones por el sistema de ejecución.

Eventos.

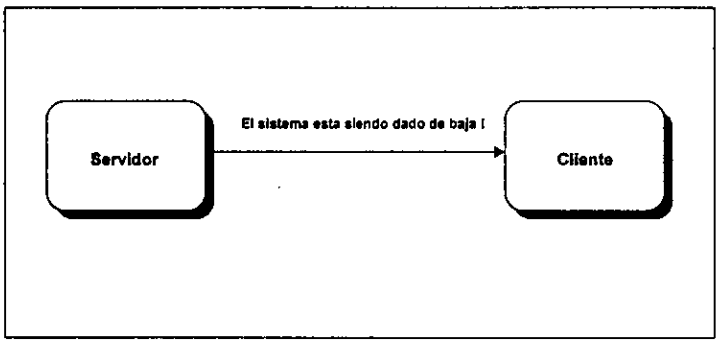
Las interacciones petición/respuesta son una forma de monitoreo (polling). Quien requiere, pide por alguna información de, o por una acción que se lleve a cabo por un servidor. El servidor responde cuando se le requiere, de otra manera, esta suspendido (idle). Algunas veces, el software no debe de monitorear, ya que el monitoreo tiende a consumir recursos y ser tedioso. En cambio, dicho software debe de ser notificado cuando la información que requiere este disponible, o en otros términos, algo ha sucedido. En este caso, el módulo que es el objetivo de la comunicación, no sabe cuando ésta será establecida, y puede, de hecho, realizar algo más mientras dicha comunicación sucede. Por ejemplo, el software que maneja una terminal, debería ser avisado cuando el sistema será dado de baja para mantenimiento, de manera que pueda avisar a la persona que está usando la terminal. De otra manera, el sistema será dado de baja, y los usuarios no tendrán pista alguna sobre lo que ha sucedido. Sin embargo, el manejador de la terminal no sólo está esperando por el sistema a ser dado de baja. Su trabajo usual es 'escuchar' por datos de entrada desde la terminal.

Llamamos a una comunicación no solicitada de esta forma, un *evento*. Un evento tiene un *generador* de eventos y *objetivos* para dichos eventos. El generador de eventos es el módulo de software que notifica que alguna condición ha sucedido y desea comunicarlo a otros módulos. Se asume que el objetivo (módulo de software) que recibe la notificación del evento, sabe que hacer una vez notificado. Si no, puede ignorar el evento o asumir que el evento es catastrófico y detener su operación.

Eventos Simples.

Con eventos simples, el generador de eventos notifica uno o más objetivos, que sabe están interesados en dicho evento. La siguiente figura ilustra el punto. Un módulo cliente es el objetivo y está siendo

informado del evento 'sistema dándose de baja', por un módulo de servidor administrativo. Un generador de eventos puede saber que objetivos notificar debido a acuerdos previos, como un conjunto estático de objetivos, o porque uno o más de dichos objetivos han informado previamente al generador de eventos, que están interesados en la notificación de un evento cuando este sea generado. En este último caso, el objetivo se 'registra' de manera directa con el generador de eventos. Esto implica que los objetivos saben cuales módulos son, de hecho, los generadores de dichos eventos.



3.2 El cliente es informado de un evento

Para notificar de los eventos a los objetivos conocidos, el generador de eventos provee de manera explícita al sistema, con información de direccionamiento para los módulos objetivo.

Por su naturaleza, los eventos son generados y notificados a los objetivos, independientemente de sí el módulo objetivo esta suspendido u ocupado procesando algo más. El módulo objetivo es interrumpido del proceso que este realizando para procesar el evento recién comunicado. Se asume que el objetivo toma nota del evento, esto, realizando probables acciones como resultado, y puede entonces continuar con el proceso que fue interrumpido. Así como no es sencillo realizar una tarea cuando se está siendo interrumpido de manera constante, la programación para el manejo de eventos, que de hecho son

interrupciones, es difícil. Una forma de manejar esto, es notificar del evento de manera sutil al objetivo, de manera que si éste último esta ocupado cuando la notificación se lleva a cabo, no sea interrumpido. Sin embargo, el evento se mantiene como pendiente, y cuando el objetivo está suspendido, entonces es notificado nuevamente. La situación es similar cuando esperamos a que alguien termine de hablar antes de hablar a dicha persona.

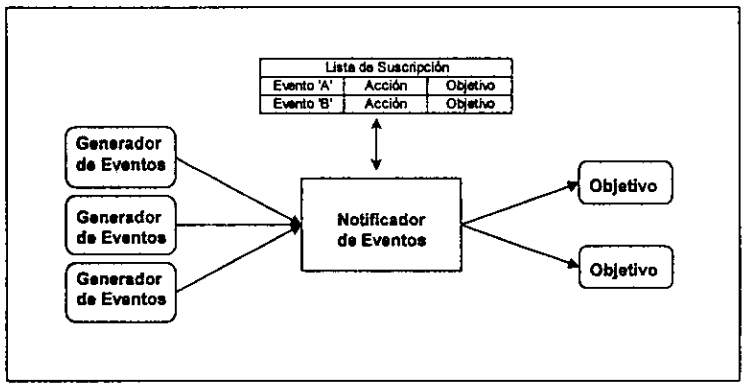
Los elementos de la comunicación de eventos simples incluyen los siguientes:
Un API para notificaciones no solicitadas, el cual es usado para enviar la notificación de los eventos a los objetivos.
Generador de eventos, que es un ejecutable, el cual envía la notificación de un evento a los objetivos que conoce.
Objetivos, son ejecutables, que son notificados cuando los eventos suceden.
Los objetivos de los eventos son conocidos por los generadores de eventos.
Datos de eventos, que pueden ser asociados con un evento en el momento en que es generado. Estos datos se ponen a disposición de los objetivos.

Eventos Notificados.

Una limitación de los eventos simples es que de alguna manera, los generadores de eventos deben de conocer los objetivos potenciales que estén interesados en la notificación de dichos sucesos. Debido a que nuevos objetivos y, nuevos generadores pueden incluirse en la aplicación, así como, objetivos y generadores pueden ser eliminados; la situación puede salir de control. Por ejemplo, una aplicación puede ser escalada, de manera que los eventos 'viejos' sean notificados por un nuevo subsistema. ¿Cómo sabrán los objetivos acerca del nuevo generador ? o viceversa. Una forma de manejar esta situación es tener un módulo de software, llamado el *notificador de eventos*, cuyo trabajo es aceptar los registros de los objetivos y notificar de los eventos. Los objetivos 'registran' su interés de ser notificados

con el notificador de eventos. Los generadores de eventos envían los eventos al notificador, el cual a su vez notifica a los objetivos registrados. Los generadores de eventos y los objetivos sólo necesitan informar al notificador, el cual hace el resto del trabajo. Por supuesto, 'hacer el resto del trabajo' puede ser una tarea compleja. Por ejemplo, un objetivo registrado puede detener su operación sin informar al notificador de dicha situación, el cual no será capaz de notificar del evento a un objetivo que no existe más.

La siguiente figura ilustra el flujo de la información a través de un notificador. Aquí, los objetivos se registran con el notificador, el cual mantiene un lista de suscriptores de tales registros. Los generadores de eventos informan al notificador cuando desean informar que un evento ha sucedido. El notificador completa las suscripciones al tomar acciones para notificar a los objetivos.



3.3 Notificación de eventos vía un Notificador

Los elementos de los eventos notificados incluyen los siguientes:

Indentificadores de eventos, son utilizados para nombrar los eventos.

Un API de eventos, el cual es utilizado para generar, suscribir, y recibir los eventos.

Generadores de eventos, que son ejecutables que indican que un evento ha sucedido.

Objetivos de eventos, son ejecutables que se registran para ser notificados cuando los eventos suceden.

Los generadores de eventos y los objetivos no se conocen.

Datos de eventos, que pueden ser asociados con un evento en el momento en que es generado. Estos datos se ponen a disposición de los objetivos.

Colas

Algunas ocasiones no es posible, requerido, o deseable, que los programas se comuniquen en línea. En lugar de ello, la comunicación entre los módulos se encola, de manera que sea procesada en algún tiempo posterior. Una analogía de dicho mecanismo ocurre cuando se deja un mensaje en una máquina contestadora de teléfono.

Para procesar posteriormente una petición entre módulos, es necesario que dejen mensajes para cada uno. Si la comunicación entre módulos se establece mientras una de las máquinas es reiniciada, será necesario dejar un mensaje almacenado de manera persistente. El almacenamiento persistente en una máquina contestadora es la cinta magnetica o disco dentro del dispositivo.

Ejemplos de uso de colas

Un uso de las colas persistentes, es almacenar datos que estaban dirigidos a un módulo de software en particular y que no estaba disponible. Quizás la máquina en donde se ejecuta dicho módulo estaba inactiva. Entonces los datos que serían la entrada de datos, pueden ser guardados en una cola persistente y reenviados cuando el módulo este disponible. Otro caso en el que las colas son útiles, es cuando los módulos que se comunican, operan a distintas velocidades. Supóngase que una computadora genera entrada de datos a otra computadora más rápido de lo que esta última puede procesar. La entrada de datos puede ser guardada ya sea en la computadora generadora o en aquella que va a procesar, hasta que la computadora que procesa pueda atender dichos datos de entrada. Claro esta, en algún punto, el proceso de mensajes debe de ser más rápido que su generación. De otra manera, las colas persistentes se llenarán.

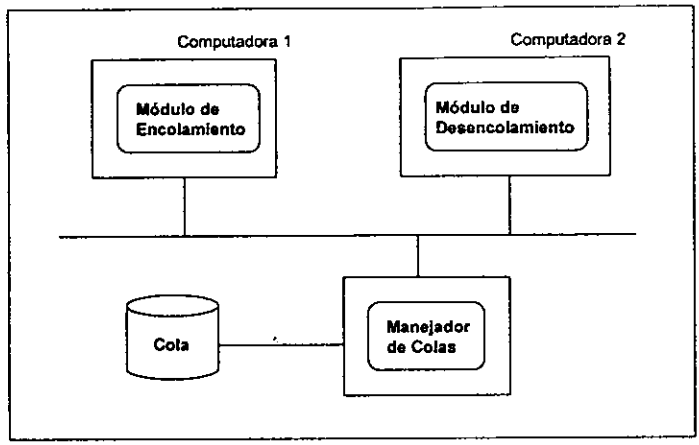
Acceso de colas

Los elementos importantes de una comunicación por colas incluyen:

- Una cola, que es el mecanismo para contener mensajes hasta que sean procesados.
- Un API de encolamiento, que provee de las interfaces para permitir a los módulos guardar (encolar) y recuperar (desencolar) mensajes de la cola.
- Módulos de encolamiento, quien guarda los mensajes en la cola.
- Módulos de desencolamiento, quien recupera los mensajes de la cola.

El propósito de una cola es realizar un proceso independiente del tiempo. La cola en sí misma es un repositorio. Así, cuando las aplicaciones se comunican vía colas, lo hacen al guardar y recuperar mensajes de la cola.

Hay que hacer notar que en un sistema distribuido, una cola, su encolamiento y desencolamiento pueden existir en distintas computadoras cada uno. Sin embargo, con la intención de simplificar las cosas para el programador, la cola debe tener independencia de locación para los módulos de encolamiento y desencolamiento. La siguiente figura ilustra un sistema de colas compuesto de un solo módulo aplicativo que encola mensajes, un módulo que desencola, y una cola, cada elemento en su computadora.



3.4 Interacción de Encolamiento

Cuando los módulos de la aplicación usan el API de encolamiento para accesar la cola, están realizando envío de datos. El envío de datos se logra a través de algún tipo de software, por ejemplo, software de distribución del sistema de archivos; el cual realiza envío de funciones para completar la tarea.

Ordenamiento en la cola

Aunque las colas son estructuras de datos que están caracterizadas por el principio FIFO (First in, First Out), hay ocasiones en que este principio debe de ser omitido. Determinada por la necesidad de modelado de las reglas de negocio, dicha omisión puede tomar una de las siguientes formas:

Mensajes por prioridad, los cuales deben de ser desencolados antes que los mensajes de menor prioridad

Respuestas y Colas

Un mensaje encolado es una petición que deberá ser procesada en algún momento en el futuro. La cola funciona como un contenedor de entrada para dichos mensajes. Sin embargo, el proceso que ha realizado el encolamiento de dicha petición, no tiene forma de saber en que momento será procesada. De hecho, cuando el proceso de la petición es realizado, el proceso quien realizó el encolamiento podría no existir más. Si el proceso del mensaje regresa resultados, puede hacerlo al colocar estos en otra cola de mensajes. Quien originó la petición (si existe todavía), puede tener acceso a la respuesta al desencolar ésta de la cola de respuestas. Si el proceso quien originó la petición no existe más, algún proceso auxiliar será el encargado de procesar las respuestas de estos procesos, cuando estas estén disponibles.

Típicamente, un módulo de encolamiento escribirá más de un mensaje en una cola. Posteriormente, verificará si hay mensajes de respuesta disponibles. La asociación de respuestas con sus peticiones es difícil, a menos que el subsistema de encolamiento, o la aplicación misma, tenga un método de correlacionar las respuestas con su correspondiente petición.

La terminología de un sistema de colas es muy similar al paradigma petición/respuesta mencionado anteriormente. La principal diferencia radica en que el paradigma petición/respuesta sucede en línea, y en el sistema de encolamiento se utiliza un mecanismo de almacenamiento persistente para poder procesar los mensajes en algún momento futuro. Además existe la independencia de identidad entre quien encola el mensaje y quien lo desencola. De igual manera que en el paradigma de Eventos Notificados, la cola es monitoreada y controlada por un agente denominado el manejador de colas (queue manager). Dicho manejador es responsable del encolamiento de los mensajes por parte de quienes son los productores de dichos mensajes y de proveer dichos mensajes a quienes los requieran vía la operación de desencolamiento.

Ya que la semántica de la petición/respuesta y el sistema de encolamiento es similar, es posible construir software que utiliza ambos paradigmas. Por ejemplo, un agente de envío, puede leer de una cola, realizar una petición/respuesta en línea, y guardar la respuesta en una cola de respuestas. El servidor cree que establece una comunicación en línea con el cliente; pero en realidad la hace a través de un agente de envío. El cliente real, obtendrá la respuesta posteriormente de una cola de respuestas.

Representación de Datos

Una parte importante de los paradigmas de comunicación, es la representación de los mensajes que son enviados entre distintos módulos de software. El posible número de tipos de estructuras de datos que pueden ser comunicados, está limitado solo por la imaginación de los arquitectos de un protocolo de aplicación. Algunos tipos genéricos son los siguientes:

- Datos arbitrarios, sin formato.
- Datos de tipo cadena de caracteres, típicamente de un conjunto de caracteres imprimibles.
- Estructuras de datos, compuestas de tipos de datos elementales de los lenguajes de programación en donde fueron originadas. Este tipo incluye, por ejemplo, registros de cobol, y estructuras de lenguaje C. La representación de las estructuras de datos requiere de un conocimiento a priori de la estructura de cada elemento por cada uno de los módulos participantes en la comunicación. Esto es, cada lado debe deberá conocer en detalle la estructura de datos.
- Datos etiquetados. Aquí, el valor de cada elemento esta acompañado por una descripción. Esto es, la colección de elementos se autodescribe. Hay que notar que debido a dicha descripción, requerirán de mayor espacio. Sin embargo, ya que contiene la propia descripción, la estructura de los elementos puede ser leída o modificada de manera dinámica. Esto provee un nivel de flexibilidad grande en la arquitectura de comunicaciones de las aplicaciones, ya que los nuevos elementos pueden ser añadidos sin afectar programas anteriores.
- Argumentos pasados a procedimientos, típicamente estos son tipos de datos elementales o tipos de datos agregados definidos en el lenguaje en el cual se ha escrito el procedimiento. Tales argumentos son posicionales, es decir, el primer argumento, el segundo, etc.

Todos los formatos mencionados anteriormente pueden ser utilizados en cualesquiera de los paradigmas mencionados hasta aquí. Ya que la comunicación sucederá entre módulos de software que se ejecutarán

en computadoras con diferente representación de los tipos de datos, por ejemplo, entre un cliente ejecutándose en un procesador SPARC y un servidor ejecutándose en un procesador pentium, las estructuras de datos deben de ser transformadas de manera que el módulo receptor las entienda. Tal proceso de transformación es denominado el servicio de presentación. Si el formato de los datos es entendible por la pila de comunicaciones, es posible para dicha pila ejecutar el servicio de presentación. Si el formato de un mensaje no es entendido por la pila de comunicaciones, entonces los módulos que se comunican, tendrán que realizar el servicio de presentación ellos mismos.

Condiciones de Error.

La parte más difícil de programar y administrar en un sistema distribuido nace del hecho mismo que el sistema es distribuido. En este tipo de sistemas, que generalmente involucran computadoras independientes, alguna falla de éstas computadoras pudieran ocasionar que parte de la aplicación se detuviera, otras partes de la aplicación pueden continuar su ejecución. Cuando las partes funcionales realizan un intento de interoperar con las partes fallidas, los escenarios de fallas suceden, y dichos escenarios son difíciles de prever tanto para el administrador como para el programador de la aplicación. Esta situación es similar, aunque en mayor escala, a cuando la conversación telefónica que estamos desarrollando es interrumpida y terminada de manera abrupta. Ninguna de las dos partes sabe que sucedió o que hacer a continuación. Cada uno tratará quizás de comunicarse con el otro, sólo para obtener un tono de ocupado. Ya que la aplicación es distribuida, sus partes, y por lo tanto, su estado, es decir el estado operacional, es distribuido.

En una aplicación distribuida, no hay lugar a donde acudir para averiguar que sucede cuando las fallas se presentan. Los tipos de error que pueden suceder son los siguientes:

- Falla de hardware. Aquí, uno de los nodos participantes en la aplicación falla.
- Falla de Red. Aquí la comunicación entre nodos, o falla o deja de existir.
- Falla de software. Aquí, o el sistema de software, o la pila de comunicaciones falla.
- Falla de combinación de hardware, red y software.

Es posible recuperarse de alguna de estas fallas. Por ejemplo, si un módulo aplicativo se detiene, puede ser posible reiniciar una instancia de dicho módulo. Si una computadora falla, se puede tener la posibilidad de alternar el proceso a un nodo de respaldo. Si la red falla, es posible dirigir el tráfico a una línea funcional, o limitar parte de la funcionalidad de la comunicación. La dificultad de convertir estas

posibilidades en realidades, es reconocer que es lo que sucede y tomar acciones correctivas. Esto se dice más fácil de lo que se hace.

Diseñando para las condiciones de error.

Debido a que existen múltiples puntos de falla en los sistemas distribuidos, el software del sistema y el de la aplicación, deben de estar bien relacionados el uno con el otro. Suponiendo un banco que quiere operar sobre la cuenta de un cliente. Hay dos categorías de falla que la aplicación podría generar:

- Falla en la semántica de la aplicación. Por ejemplo, suponiendo que el cliente tiene un balance de cero, y quiere hacer un retiro de la cuenta, y el banco no permite el sobregiro. Aquí, las reglas del negocio han sido violadas, y el proceso no debe de continuar. La aplicación ejecuta correctamente al no terminar la operación.
- Falla de comunicación. Algo sucedió mientras se ejecutaba la transacción. Quizá el servidor no esta más en ejecución. Aquí el sistema no puede ejecutar una operación que es permitida por las reglas del negocio.

Usualmente, estas dos modalidades de falla son independientes. Las reglas de negocio de la aplicación son independientes de las comunicaciones y viceversa. Sin embargo es importante lograr que ambos tipos de fallas sean indicados al sistema. Por ejemplo, en el paradigma petición/respuesta, debe permitirse un indicador de éxito/fracaso, así como el indicador éxito/fracaso de la comunicación.

Timeouts

Muchas fallas resultarán en una espera indeterminada, esto es, esperando por un mensaje de respuesta que, o no llegará, o no puede ser entregado. Esto es difícil de manejar para el programador. Una forma de ayudar a controlar este problema, es que el API de comunicación ofrezca la noción de time-out. Dada esta noción, las peticiones que esperan una respuesta son satisfechas, ya sea por el mensaje esperado, o por el sistema al indicar que el límite de tiempo de espera ha sido excedido. En el último caso, hay noticias buenas y malas. La noticia buena es que la aplicación no esperará por siempre; la noticia mala es que no se puede saber con certeza que ha sucedido. Hay que notar que un time-out se considera como error de sistema en lugar de un error de semántica de la aplicación. Claro esta, un mensaje puede llegar aún cuando la señal de time-out ha sido enviada. Aquí, la aplicación cree que no hay mensaje de regreso, pero, de hecho, uno ha llegado. Para proveer una vista consistente de la aplicación, la mejor opción para el API de comunicaciones es descartar dichos mensajes.

Transacciones

Cuando una aplicación encuentra un error, hay ocasiones en que sería recomendable iniciar de nuevo la operación, u olvidarse de ésta. Las transacciones proveen lo anterior, al proveer a la aplicación de una herramienta para deshacer los efectos de una operación que previamente se ejecutó. Hay que notar que la operación como tal, no puede ser desecha, pero los resultados de ésta, sí. Por ejemplo, las actualizaciones a la base de datos llevadas a cabo durante la ejecución de la operación pueden ser sometidas a la operación de un rollback, de manera que los valores afectados regresen a su estado original.

Noción de Transacción

La propiedad más importante de una transacción para un programador, es la atomicidad. La idea es la siguiente. Una aplicación aisla un conjunto de instrucciones por medio de llamadas que definen los límites de la transacción, tales como begin_tran y end_tran. Cualesquiera situación que suceda, todos los efectos de la ejecución del conjunto de instrucciones aisladas, o en su caso, ningún efecto; deben de ser permanentes. Esto permite que un número de operaciones sean ejecutadas, como una sola operación. Esto es lo que significa atomicidad.

La selección de qué acciones deben de ser hechas de manera atómica es parte de la aplicación. En efecto, representan algunas de las reglas del negocio a ser programadas. Por ejemplo, para una operación de transferencia, la regla sería "cualquier operación de transferencia debe de incluir una operación de retiro exitosa y una operación de depósito, exitosa también". Así, es la aplicación quien conoce donde deben de iniciarse y terminarse las transacciones; y es responsabilidad el programador establecer dichas transacciones.

Ayuda de las transacciones

El principal valor de las transacciones para un programador es que aseguran la atomicidad de las operaciones que conforman una regla de negocio. Sin soporte transaccional, un programador debe de proveer de la lógica necesaria para reforzar las reglas del negocio. Por ejemplo, en el caso previo de la transferencia de fondos, si el hardware falla cuando una de las operaciones ha sido completada, las bases de datos quedan en un estado inconsistente. Cuando las operaciones son restauradas, una acción correctiva debe de ser iniciada con el objetivo de mantener la consistencia de los datos. Tales acciones son denominadas transacciones de compensación. Dichas transacciones son difíciles de programar, ya que debe de estar disponible en algún lado, el estado del progreso de la función y la suficiente información para completar o deshacer la operación. Más aún, la ejecución de una transacción de compensación, puede ser interrumpida también, requiriendo de otra transacción de compensación. Así, la aplicación debe de asegurar en todos los puntos, que hay suficiente información para terminar la operación o someterla a la operación de rollback. Para las transacciones complejas que involucran varias operaciones, quizás entre múltiples sistemas de computo, las transacciones de compensación pueden ser muy complejas y susceptibles de error. De hecho, el programador tiene que desarrollar un sistema de transacciones.

Transacciones en un ambiente distribuido

¿ Qué significan las transacciones en un ambiente distribuido ? Cuando una aplicación comienza una transacción, está indicando, que hasta que indique que la operación esta completa, las operaciones a realizar deben de ser llevadas a cabo de manera atómica. Cuando un módulo aplicativo que ha comenzado una transacción se comunica con un nodo remoto que es también parte de la aplicación, el trabajo que lleve a cabo este último módulo debe de ser parte de la transacción. Esto es, todo el trabajo, ya sea local o remoto, está sometido a las mismas propiedades atómicas. Si se da el caso de que el nodo

remoto se comunique con otro nodo remoto más, las operaciones involucradas por dicho nodo deberán ser incluidas dentro de las operaciones de la transacción en progreso. Este tipo de comportamiento es bueno para la aplicación, ya que si alguna de las tantas operaciones involucradas falla, la transacción restaura todo a su estado original antes de haber iniciado la transacción.

Las transacciones no necesariamente tienen que abarcar las comunicaciones. Por ejemplo, algunas comunicaciones pueden ser de naturaleza informativa, y su éxito puede no ser importante del todo. En otros casos, ciertos protocolos de información o semántica de la aplicación pueden no requerir de control transaccional.

Manejadores de recursos

Previamente, se menciono que no son las operaciones de una ejecución las que se deshacen, sino los efectos de dichas operaciones. ¿ Qué significa esto ?

La ejecución de una aplicación cambia su estado. El estado está representado por el contenido en la memoria de la computadora, la permanente y la temporal. En general, la ejecución de una transacción cambia la memoria permanente de una aplicación. Por ejemplo, un retiro de una cuenta bancaria, cambia el monto que de manera permanente mantiene la aplicación, hasta que un nuevo cambio sucede.

El estado permanente de una aplicación es realizado en la memoria de almacenamiento que ocupa. Esta memoria permanente está generalmente organizada en archivos y bases de datos. Estos mecanismos de almacenamiento son conocidos como recursos de la aplicación, y son accesados vía un manejador de recursos (resource manager) o RM. El acceso a la información guardada por un RM, es mediante el API del manejador de recursos. Dicho API incluye interfaces para el manejo de archivos, manejo de registros,

y manejo de sistemas relacionales de datos (RDBMS), siendo la interfase estándar para este último, el SQL.

Es importante hacer notar que una aplicación puede accesar varios RM's cuando ejecuta una operación del negocio. Por ejemplo, una cuenta de cheques puede usar un RM para los retiros y otro RM para los depósitos. Para soportar las operaciones complejas del negocio, los RM's son por naturaleza transaccionales. Las transacciones provistas por dichos RM's son internas, y son delimitadas por funciones que son provistas como parte del API del RM. En general, los RM's con capaces de proveer atomicidad.

Las aplicaciones distribuidas, sin embargo, utilizan varios RM para poder completar una transacción. De hecho, una aplicación distribuida bien diseñada, no sabe de que manera se realizan las operaciones remotas. En particular, el RM utilizado por el servidor para llevar a cabo la tarea, no debe se ser conocido por el cliente. Esto permite que la implantación del servidor pueda cambiar, y que posiblemente pueda usar un RM mejorado. Cuando una aplicación utiliza múltiples RM's, existe la necesidad de proveer funcionalidad combinada entre las múltiples transacciones que inician dichos RM's. Tal funcionalidad es conocida como una transacción global.

Manejadores de transacciones globales

El hecho de que las operaciones de nodos remotos sean incluidas como parte de la transacción en progreso, no es suficiente para soportar la noción de transacción. Alguien debe de recordar qué partes de la aplicación están involucradas en una transacción, y de indicarles cuando la aplicación ha terminado ésta, ya sea para completar el trabajo o para deshacerlo. Este 'alguien' es llamado un manejador de transacciones (Transaction Manager) o TM. Los TM's son sistemas de software que propagan las transacciones cuando se establece comunicación entre módulos, y coordinan la ejecución de la

transacción, ya sea que ésta sea exitosa o fallida. Ya que los RM's mantienen el estado de los datos, estos juegan un papel importante en las transacciones globales; un TM es responsable de lo siguiente:

- Aceptar la petición de la aplicación para iniciar y terminar transacciones. Esta funcionalidad esta dada por un API para delimitar las transacciones.
- Propagar la transacción cuando hay comunicación entre módulos.
- Hacer interface con los RM's para informarles que la transacción global es operacional e instruirles durante la fase de terminación o recuperación de transacciones.

En un esquema donde el TM funge como coordinador de transacciones, y el RM como subordinado; un RM es responsable de lo siguiente:

- Aceptar comandos de delimitación de la transacción por parte del TM.
- Realizar el trabajo en nombre de la transacción. Típicamente, el RM provee de las propiedades de consistencia, aislamiento, y durabilidad de la transacción.
- Aceptar comandos para realizar las operaciones para completar la transacción, que son indicados por el TM.

Para hacer una analogía, supóngase una orquesta de música. Un TM es el director de la orquesta. Dicho elemento tiene la visión general, incluyendo los elementos y partes que cada RM – los músicos – tienen que ejecutar. Los TMs dirigen a los RMs durante el curso de la transacción, y más importante aún, durante la etapa de conclusión de esta.

Transacciones y colas

Previamente se ha mencionado el paradigma de las colas persistentes para comunicaciones diferidas. Las colas son sistemas de almacenamiento, a través de los cuales, los componentes de una aplicación se pueden comunicar de manera diferida. Las colas son pasivas por naturaleza, aceptan peticiones para guardar o borrar paquetes, pero nunca inician tales acciones. Una cola, es en efecto un RM especializado, y el mecanismo de almacenamiento para las colas es llamado 'cola base'. Las peticiones para proceso futuro, esto es, los cambios de estado a la aplicación, pueden ser guardados en colas, y posteriormente ser extraídos. Las colas mismas son parte del estado de la aplicación. Ya que las colas son RMs, son transaccionales por naturaleza. Esto significa que encolar una operación esta contenida dentro de una transacción. Si la operación no se completa, la operación de encolamiento es abortada. Esto es, una vez que se realiza el rollback, el paquete es removido de la cola. Otro manera de decir esto es, los mensajes encolados aparecen en una cola, y así, están disponibles para un desencolamiento posterior, sólo hasta que la transacción que contienen hace commit. De igual manera, una operación de desencolamiento que se ve involucrada en una operación de rollback, deja el mensaje en la cola, así, disponible para un re-proceso después de un período de tiempo.

El concepto de las colas transaccionales es muy útil, ya que habilita la construcción de un conjunto de transacciones en el tiempo.

El resultado de las colas transacciones puede ser la entrada a un filtro. Un filtro es un módulo de la aplicación que se encarga de ver el seguimiento de las reglas del negocio. Ya que las colas están sujetas a las transacciones, la ejecución de cada filtro, es un punto susceptible de falla. Si un filtro falla durante la ejecución, la entrada será recuperada a su estado original, y la salida eliminada, así como cualquier efecto colateral que se hayan producido, por ejemplo, actualizaciones a bases de datos, borrado. Esto permite dejar el filtro listo para una nueva ejecución.

Administración de una Aplicación

Se han descrito técnicas y métodos de comunicación que permiten la construcción de aplicaciones distribuidas. Tales aplicaciones tienen el objetivo de automatizar la operación de una empresa. Después de construir y probar la aplicación, la empresa implanta dicha aplicación. En este punto, otra persona, distinta a quien creó dicha aplicación, pondrá el software en operación y asegurará su operación de manera satisfactoria.

Muchas organizaciones están conscientes de los costos para desarrollar una aplicación. Pero pocas, están conscientes de los costos para mantener dicha aplicación o sistema en operación. La administración es requerida para asegurar que las funciones del negocio que la aplicación automatiza sean ejecutadas de manera adecuada y en tiempo, y que los usuarios de la aplicación, incluyendo a los empleados mismos, puedan recibir los servicios, que la aplicación esta destinada a proveer.

El objetivo, es lograr la administración y manejo de las aplicaciones distribuidas. Para lograr esto, las aplicaciones y software sobre las cuales la aplicación ha sido construida, necesitan ser conocidos, de manera que permitan implantar herramientas de administración.

Tareas administrativas

Típicamente, la administración de una aplicación incluye lo siguiente:

- Instalar el software de la aplicación en las computadoras donde será utilizado.
- Establecer los parámetros de configuración para adaptar la aplicación en el ambiente de trabajo.
- Configurar el ambiente, como el sistema operativo, para soportar la ejecución de la aplicación.

- Configurar la aplicación de manera tal, que aproveche el ambiente al máximo.
- Inicio y terminación de partes de la aplicación, como sea necesario.
- Monitoreo de la ejecución de la aplicación para asegurar los servicios a los clientes.
- Respuesta a las alarmas, incluyendo atentados de violación de seguridad, que la aplicación genere mientras se encuentra en ejecución.
- Actualizar la aplicación con versiones que incluyan mejoras de funcionalidad o que añadan nuevos componentes.
- Instalación de 'parches de software' de manera que se corrijan errores de implantación de la aplicación.

La administración de una aplicación distribuida es mucho más compleja, que una aplicación monolítica. Ya que la aplicación distribuida está particionada, algunas partes fallaran, mientras otras, se mantendrán en operación.

La administración de una aplicación distribuida, puede ser distribuida o centralizada. La selección depende entre otras cosas del alcance geográfico de la aplicación, su tamaño y muchos otros factores. Ciertamente es deseable poder administrar una aplicación distribuida desde un sólo lugar. La administración de una aplicación distribuida, requiere que el software de administración se encuentre instalado en las máquinas en donde la aplicación se encuentra. Sin embargo, la interfase humana, para dicho software de administración, puede ser manejada desde una sola consola administrativa. Tal consola, permite al operador interrogar a los componentes de la aplicación, recibir alarmas de estos, y reconfigurar y operar la aplicación, vía varios comandos administrativos.

Además de permitir la administración desde una consola central, otras facilidades se requieren para facilitar el trabajo de administración:

- Interfaces de programación estandarizadas para los datos administrativos del esquema distribuido de la aplicación.
- Presentación y modificación de los datos administrativos a través de interfaces amigables (GUIs).
- Facilidad para permitir que las aplicaciones permitan una interfase administrativa.

Definición de la aplicación

La siguiente definición de 'aplicación' es informal. Una aplicación es el software y recursos permanentes (bases de datos, archivos, RMs, etc...) utilizados para automatizar una o varias funciones del negocio. Una aplicación distribuida, es aquella que se ejecuta en múltiples computadoras. Las aplicaciones crecen al añadir nueva funcionalidad. Esto se logra, al añadir nuevo software y posiblemente, nuevos recursos. De tal manera que los límites de la aplicación sean conocidos, es útil definir la aplicación a través de un sistema de configuración. Típicamente, esto identifica el código, los datos, y los recursos de computadora que comprende la aplicación. Una aplicación se puede comunicar con otra aplicación. En tal caso, se visualiza la unión de dos aplicaciones, como una gran aplicación. Sin embargo, de manera implícita en la noción de la aplicación, esta el hecho de que pueda ser manejada y administrada como una entidad. Así, hay que diferenciar entre dos aplicaciones que necesitan interactuar, y las partes de una aplicación que interactuan. El primer caso representa dos aplicaciones que se administran de manera independiente. El último caso, representa una entidad administrable.

Por ejemplo, supóngase una compañía que tiene gente de ventas, que son pagados parte en comisión y parte con un promedio de horas trabajadas. La compañía tiene dos sistemas, una para la nómina general y uno para el pago al departamento de ventas. El propósito principal del sistema de nómina general, es automatizar el pago a los empleados de la compañía. El propósito del sistema de pago al departamento de ventas es llevar un registro de las ventas hechas y potenciales. Cada sistema es administrado de manera independiente. Para compensar a la gente de ventas, con base en el horario trabajado, el

sistema de nómina general recibe los datos de las tarjetas checadoras que el departamento de ventas entrega. Para compensar a los vendedores por sus ventas, el sistema de pago del departamento de ventas, necesita informar de las ventas netas que cada vendedor logró, al sistema de nómina general. Así, dentro de la misma compañía, hay dos sistemas independientes que necesitan establecer comunicación.

Administadores y Entidades Administradas

Se mencionó ya que uno de los retos de las aplicaciones distribuidas es su manejo operacional. Esto incluye tanto el hardware en el cual se ejecuta como los componentes de software. Un determinado grupo de estándares y productos, por ejemplo, OpenView de HP, ha surgido en el mercado para poder proveer de un esquema de administración, en el cual, los sistemas distribuidos y las aplicaciones, conocidas como entidades administradas, pueden ser manejadas. En general, el papel de las entidades administradas es:

- Aceptar comandos del administrador para reportar datos de la operación.
- Aceptar instrucciones del administrador para modificar la operación.
- Informar al administrador de un comportamiento extraño.

Para lograr dicho propósito, el administrador, necesita recuperar información y actualizar información de la aplicación. Esto se puede lograr mediante agentes. Dichos agentes, que se encuentran entre la aplicación y el software de administración, aceptan las peticiones de recuperar o actualizar información del administrador, los traduce en operaciones o instrucciones que la aplicación ejecuta y reporta de regreso sobre el resultado de dichas operaciones al administrador.

Seguridad

Las aplicaciones centralizadas son más fáciles de proteger contra la violación de seguridad que las distribuidas. Estas últimas ofrecen más puntos vulnerables para poder interceptar datos, interrumpir operaciones, o generar entrada de datos fraudulentos. Así como las aplicaciones distribuidas se vuelven cada vez más complejas, deben de estar protegidas de este tipo de acceso invalido. Algunos aspectos de seguridad pueden y deben de ser provistos por la infraestructura con la que se cuente para implantar la aplicación. Por ejemplo, los mensajes entre módulos de la aplicación podrían ser encriptados para asegurar la privacía de estos. Por otro lado, algunos aspectos de seguridad son específicos de la aplicación. Por ejemplo, un banco puede tener la política en la que un retiro considerable requiere de la autorización de un gerente y la aplicación debe reforzar dicha política.

Ningún sistema es 100% seguro, partiendo del hecho de que los usuarios mismos del sistema pueden, de manera inadvertida o a propósito comprometer la seguridad de dicho sistema. Sin embargo, es razonable que un negocio espere que el software que utiliza para implantar su sistema distribuido, otorgue facilidades de seguridad genéricas para proteger la operación de sus sistema.

El tópico de la seguridad computacional es grande y complicado, y no es propósito de esta tesis hacer un estudio de dicho tema. Sin embargo se ha de mencionar las principales áreas de seguridad, que son: Autenticación, Autorización, Privacía, y Auditoría.

La Autenticación es el acto de asegurar que el usuario del sistema sea quien dice ser. Además de que el sistema autentica a los usuarios, estos últimos autentican el sistema. Esto es, que necesitan estar seguros, de que cuando se firma a un sistema, es el sistema real y no, uno falso. Cuando ambos lados determinan que el otro es quien dice ser, se dice que las partes se han autenticado mutuamente. Los

usuario autenticados se denominan principales. Típicamente, un usuario prueba su identidad al revelar algún secreto, como un password, compartido con el sistema.

La Autorización significa el permitir o no permitir a los principales accesar los recursos del sistema. Por ejemplo, solo el principal cuyo nombre es 'administrador' puede leer y escribir un archivo de password. Un ACL es una lista de acceso de control que determina que principal tiene acceso a que recursos.

La Privacía significa asegurar que los mensajes enviados entre programas no sean entendidos por usuarios no autorizados. La privacia de los mensajes, generalmente se logra al encriptarlos antes de su envío, y al desencriptarlos cuando se reciben.

La Auditoría significa el registro del acceso al sistema. Tal registro puede ser utilizado posteriormente para determinar las acciones que el usuario llevó a cabo en el sistema, y si estas acciones modificaron datos.

Capitulo 4

Este capitulo describe la arquitectura del prototipo y las características funcionales de este. Se mencionan varias posibilidades de expansión de funcionalidad para implantar un sistema real.

Arquitectura del Prototipo

Componentes

El prototipo está formado de tres componentes. El cliente – el primer componente de la arquitectura - al hacer peticiones invoca componentes de servidores que son administrables. El componente de servidores implanta y publica las funciones del negocio como servicios disponibles a quienes los requieran, éstos pueden estar ejecutandose en un ambiente de procesos distribuidos. La Base de Datos constituye el tercer componente y administra los recursos de información del prototipo.

Las aplicaciones de negocios criticas son generalmente transaccionales y deben tener un desempeño exacto de integridad y de administración. Estos requerimientos determinan el enfoque arquitectónico a seguir para el desarrollo de la aplicación, instalación y operación. El prototipo presentado en esta tesis provee una solución a este tipo de aplicaciones en ambientes distribuidos.

Muchas organizaciones reconocen ahora a las aplicaciones basadas en componentes como una evolución lógica de sus esfuerzos por construir aplicaciones distribuidas.

Los sistemas basados en Mainframe son ahora mejorados por una larga base de sistemas de servidores y de escritorio. Estos sistemas distribuidos son acoplados a nivel de red por transportes estándar, y forman un recurso de conexión. Inicialmente esta infraestructura sirvió para migrar las aplicaciones de oficina desde los sistemas centralizados – principalmente los procesadores de documentos y comunicaciones de correo electrónico que son fácilmente implantadas en PC's y servidores de archivos.

También aplicaciones de bases de datos cliente/servidor de dos capas que utilizaron el ambiente distribuido fueron implantadas a nivel departamental. Estas primeras aplicaciones cliente/servidor proveen

el concepto para aplicaciones de proceso distribuido, pero permanecen limitadas en cuanto a su escalabilidad y administración. Más importante aún, el tipo de acceso a los datos dejaba a la aplicación en un estado monolítico e incapaz de utilizar los recursos de red apropiamente.

Los componentes mejoran la arquitectura cliente/servidor, al soportar la partición de aplicaciones y permitir el desarrollo e implantación de la lógica del negocio de una manera más efectiva, así como administrar su ejecución en un ambiente de red. La utilización de componentes promueve una separación limpia y funcional:

- presentación/interacción con el usuario,
- lógica del negocio parte central del sistema,
- manejo de datos.

Esta separación beneficia el desarrollo de la aplicación, y soporta su ejecución óptima en un ambiente distribuido.

Se pueden obtener dos beneficios al separar la presentación y la lógica del negocio - que crece proporcionalmente con el número de clientes que participan en la aplicación -. Primero, el desarrollo de componentes permite a los diseñadores de la aplicación concentrarse en la encapsulación de las funciones del negocio, el flujo de datos y la interacción con el usuario. Al concentrarse en este punto, los diseñadores pueden rápidamente hacer prototipos de los procesos del negocio y de los elementos funcionales claves. Las consideraciones de presentación, aunque son importantes, son subordinadas a la lógica del negocio.

Segundo, La separación de la presentación y la lógica del negocio, soporta durante el ciclo de vida productivo una buena administración del código de la aplicación. La distribución frecuente de actualizaciones de aplicaciones de escritorio – en algunas ocasiones a cientos de usuarios -, es un proceso de mantenimiento costoso y lento para muchas organizaciones. Con el uso de componentes los

elementos de presentación (entradas de datos y despliegue de información), son separados del funcionamiento interno y de la lógica del negocio. Cambios en la lógica del negocio, encapsulada en los servicios ofrecidos, no requerirán de cambios en las aplicaciones de los usuarios. Por ejemplo: si las politicas para la autorización de otorgamiento de créditos dentro de una aplicación bancaria cambian, las modificaciones requeridas se realizaran dentro del componente servidor y no involucrarán un manteniento a las aplicaciones de los usuarios. Adicionalmente, los cambios en el modelo de datos pueden requerir de cambios en los servicios que accesan a estos, generalmente ésto puede ser hecho de manera transparente ya que el esquema de la base de datos esta separado de la lógica del cliente.

Al separar el componente servidor de la capa de presentación y de la capa de datos permite al prototipo ser administrable de una manera óptima. Lo anterior se logra al configurar los componentes, y administrar el flujo de los mensajes y de las peticiones. Por ejemplo, dentro del prototipo se pueden iniciar y terminar servidores, replicarlos en respuesta a la demanda, habilitar/deshabilitar servicios de manera dinámica. Esta capacidad de administración añade escalabilidad y confiabilidad a la aplicación.

Requerimientos para Aplicaciones Basadas en Componentes

Una aplicación distribuida puede ser particionada de muchas maneras, sin embargo, un particionamiento óptimo esta basado en el proceso que involucra al negocio, los recursos de hardware disponibles para la aplicación, el tipo y ubicación de los recursos de datos. Los diseñadores de la aplicación deben contar con la flexibilidad de poder particionarla funcionalmente en componentes, así como de instalar éstos últimos en configuraciones de hardware heterogéneas. De tal manera, que si los recursos son cambiados, también el particionamiento de componentes y la instalación de estos lo hará.

Un conjunto rico y confiable de mecanismos de comunicación/mensaje para la aplicación. El sistema de comunicaciones debe de ser de alto nivel – independiente de los distintos protocolos sobre los cuales esté montado – y proveer de un buen conjunto de comunicaciones. Para una máxima flexibilidad en el diseño de la aplicación, deben ser soportados: mecánismos de comunicación síncronos, asíncronos, conversacionales, basados en eventos y de almacena-envia. Los diseñadores de la aplicación deben de tener la habilidad de utilizar estos recursos dentro un de un conjunto de componentes determinado.

Manejo dinámico de la aplicación. Las aplicaciones se ejecutan en un ambiente dinámico y cambiante. La carga varía constantemente, los recursos de cómputo entran y salen del proceso, e inevitablemente las fallas pueden presentarse. Un ambiente manejable de componentes de servidores debe soportar una configuración dinámica. Por ejemplo, es posible que se requiera aumentar el numero de servidores o disminuirlos en base al aumento o disminución de la carga de trabajo.

Administración del ambiente de una aplicación distribuida. Por naturaleza, las aplicaciones distribuidas requieren de vigilancia, seguridad, monitoreo de desempeño, revisión de fallas, manejo de alertas de limite de recursos y una variedad de operaciones más que se complican en un ambiente como este. Se debe tener capacidad de administración superior para tener una aplicación distribuida en línea y mantenerlo así.

Panorama General del Modelo de Componentes del Prototipo

La arquitectura del prototipo esta compuesta de tres capas:

Clientes:

- La función primaria del cliente es obtener información para un proceso y desplegar información. El fácil uso de la aplicación y el uso intuitivo generalmente va ligado a las pantallas, el orden y los métodos con los cuales la información es obtenida y desplegada.
- Las funciones centrales del negocio son accesadas como "servicios nombrados" disponibles al cliente; el cliente envía una petición a la aplicación de servicios correspondiente por medio de un mensaje.
- El cliente recibe y procesa las respuestas del componente servidor.
- En aplicaciones grandes y escalables, muchos clientes se conectan de manera concurrente y comparten un conjunto de servidores, así como los servicio encapsulados en estos últimos.

Servidores:

- Los servidores encapsulan las funciones centrales del negocio en forma de "servicios nombrados". Estos servicios son generalmente funciones y están dedicados a una sola tarea (ej. Obtener número de cuenta de un cliente).
- Los servicios encapsulan el acceso de los datos (por ejemplo, al incluir ESQL inmerso) o lógica de proceso para base de datos; los clientes requieren actualización por medio de la petición a un servicio, y éste último es el que realiza la interacción con la base de datos.
- El componente de servidor puede hacer disponibles muchas rutinas de servicios individuales, de las cuales cada una puede ser invocada por nombre desde los clientes autorizados.

- Una aplicación distribuida típicamente incluye uno o más servidores, y cada uno de ellos agrupa las rutinas de servicio para un determinado grupo de acciones o tareas que identifican una función del negocio.
- Cualquier numero de clientes, concurrentes y autorizados, puede realizar peticiones. Una función clave de esta capa es poder atender estas peticiones a los servicios basándose en: prioridades, carga de trabajo de los servidores y de los recursos disponibles en el momento.

Datos:

- Los datos son almacenados, usualmente, en bases de datos, sistemas de archivos indexados, y mecanismos de colas de almacenamiento-envio.
- El componente de servicios de la aplicación actúa como cliente para estos servidores de datos al enviar SQL, peticiones de encolamiento/desencolamiento, y otros protocolos de acceso a los datos para el administrador de recursos específico.

El mecanismo de mensajes utilizado en este prototipo permite a los clientes localizar e invocar servicios dentro de un conjunto de servidores instalados dentro de la red. La ejecución de esta aplicación requiere invocar uno o más servicios para completar el proceso de la transacción. El mismo mecanismo de mensajes que se emplea para conectar el cliente con el servidor, también es utilizado en las comunicaciones entre servidores (cuando alguno de estos actúa como un cliente al hacer una petición a otro servidor). La clave para el diseño de este prototipo con una arquitectura de tres capas, es la división de la aplicación en diversos componentes.

El prototipo soporta facilidades de administración requeridas para la confiabilidad, integridad y afinamiento de su desempeño, incluyendo lo siguiente:

- Bases de Manejo de Información (MIB por sus siglas en inglés – Management Information Bases) para el almacenamiento de información crítica del sistema.
- Un API Administrativo para poder programar acceso a las tablas centrales de ambiente y actualización del sistema.
- Un mecanismo de alerta de eventos para notificación inmediata de eventos del sistema o de la aplicación.
- Servicios de administración de la aplicación para poder definir componentes de servidores y servicios, así como la administración dinámica de su ejecución.

Vista General de la Arquitectura de Componentes del Prototipo

El prototipo llena los requerimientos de un cómputo efectivo basado en componentes. Contiene los elementos como: componentes de aplicaciones tipo cliente/servidor y servicios distribuidos, que son necesarios para su manejo y administración de manera confiable en un ambiente heterogéneo. La flexibilidad del prototipo se deriva de su propia arquitectura cliente/servidor y su modelo de proceso.

Las aplicaciones de componentes, como este prototipo, se ejecutan como un grupo de procesos que cooperan utilizando comunicaciones basadas en mensajes. Por ejemplo, una petición a una aplicación de tres capas cliente/servidor involucra los siguientes pasos:

1. Un proceso cliente típicamente una aplicación de escritorio llama a un servicio.
2. Un mensaje de petición es construido y viaje a través de la red hasta un componente servidor que incluye el identificador del cliente.
3. Como parte de esta comunicación el proceso administrador de los recursos de la aplicación es accesado para seleccionar un servidor que contiene el servicio invocado, regresando la dirección de la cola de mensajes al identificador del cliente.
4. El mensaje de petición es entonces transferido del identificador del cliente a la cola de mensajes del servidor para su proceso.
5. El servidor lee el mensaje de su cola e invoca el nombre del servicio requerido; típicamente un servicio accesa una base de datos, quizá realizando una actualización. El manejador de recursos de datos puede estar localizado en el mismo nodo donde se encuentran los servidores/servicios u ejecutándose en un servidor remoto. En un sentido estricto, un servicio es un cliente de una manejador de recursos de datos.

6. Una vez realizada la tarea del servicio, el menaje de respuesta es colocado en la cola de mensajes del cliente, el cual regresara al proceso que lo haya invocado inicialmente.
7. Por definición después de procesar una petición el servidor regresa a su estado inicial y esta listo para procesar la siguiente petición en su cola de mensajes.

Ya que el prototipo maneja dinámicamente las peticiones a los servidores, les permite un control de estas peticiones por medio del uso de la información localidad en el proceso administrador de los recursos de la aplicación, para realizar balanceo de cargas de trabajo entre los servidores existentes. Lo anterior significa que se puede configurar la relación entre clientes y servidores de la siguiente manera:

1. Muchos clientes trabajando con un servidor.
2. Muchos clientes trabajando con muchos servidores.
3. Un cliente trabajando con muchos servidores.

Utilizando el mutiplexaje y las diversas relaciones entre servidores y clientes, el prototipo contiene las capacidades asociadas con aplicaciones críticas de negocios:

1. Capacidad para una gran número de usuarios concurrentes.
2. Capacidad para un gran volumen de datos centralizados o distribuidos.
3. Gran capacidad de manejo de mensajes.
4. Tiempo de respuestas cortos y predecibles.
5. Integridad y seguridad de datos.
6. Alta disponibilidad, incluyendo procesos de 7 x 24.

Funcionalidad

La funcionalidad del prototipo puede variar, ya que puede estar configurado de las siguientes maneras: un solo servidor y pocos clientes, como una aplicación distribuida de gran escala con miles de clientes, cientos de servidores con un largo conjunto de componentes de servidores y servicios.

El prototipo está especificado por archivos de configuración que se traducen en un conjunto de rutinas de bases de información compartidas. Estas bases compartidas residen en cada nodo participante.

El prototipo contiene un manejador de transacciones (Manejador de Transacciones) , que provee de las estructuras de datos y servicios para un proceso dinámico de la aplicación. Este manejador de transacciones esta soportado por un grupo de subsistemas que proveen funcionalidad distribuida avanzada en las áreas de manejo de clientes, conectividad a otros nodos y configuración de la aplicación. Estos subsistemas son los siguientes:

1. Manejador de transacciones (Manejador de Transacciones).
2. Workstations.
3. Dominios.
4. Servicios de encolamiento.

Workstations

Es un proceso con capacidad de multiplexaje conocido como el Workstations handler, reside en la aplicación para poder manejar las comunicaciones entre clientes Workstations y el Manejador de Transacciones.

Dominios

Esta es una característica que extiende el alcance del modelo cliente/servidor, que permite al prototipo poder compartir los servicios con otras aplicaciones autónomas. Un dominio es un conjunto de servicios en un ambiente configurado.

Cuando el prototipo llegue a crecer de manera tal que incluya múltiples procesos del negocio, entonces la partición de la aplicación es adecuada. La partición por dominios crea múltiples configuraciones, como la del prototipo, que permiten la creación de aplicaciones autónomas.

Dominios provee del mecanismo para administrar el prototipo permitiendole la integración con múltiples aplicaciones de una manera transparente, y además permite una administración autónoma en cada aplicación participante.

Este prototipo es una aplicación totalmente autónoma – administrada de manera independiente de otras aplicaciones. Por esta razón, es llamada domino. El alcance de las tareas de administración es un factor primario en la arquitectura de una aplicación distribuida. Otros factores podría ser el tamaño de la aplicación, la organización de las políticas dentro de la empresa, la confiabilidad de los límites, y los grupos de usuarios que necesitan acceso a recursos y servicios compartidos.

Este prototipo puede cooperar con otras aplicaciones. Esta cooperación consiste de especificar que servicios son accesibles entre las aplicaciones. Una aplicación exporta información acerca de como accesar un conjunto de servicios aplicativos, y otra aplicación importa esta información.

La facilidad de cooperación entre aplicaciones consiste de un conjunto de herramientas administrativas que permiten a los administradores especificar que servicios son accesibles a traves de las aplicaciones.

Servicios de encolamiento

El prototipo contiene de un mecanismo simple para encolar y desencolar peticiones y respuestas. Los servicios de encolamiento permiten lo siguiente:

- Entrega garantizada de transacciones.
- Entrega de peticiones por tiempo.
- Control transaccional para el encolamiento de las peticiones.

Este mecanismo del prototipo le permite poder llevar a cabo transacciones en modo "batch" o por determinados periodos de tiempo.

Al momento de desarrollar la aplicación no fue necesario saber donde iban a ser ubicadas estas colas aplicativas. Esta transparencia permite a los administradores de la aplicación mover las colas aplicativas de una máquina a otra sin tener que modificar el código original.

Configuración de la Aplicación

El prototipo incluye los recursos requeridos para ejecutarse en un ambiente distribuido:

- Recursos que incluyen información perteneciente a los atributos globales del prototipo, como niveles de seguridad, balanceo de cargas, y la definición de recursos específicos para activar la aplicación y recuperarla en caso de falla.
- Definición de cada maquina participante en la aplicación y la especificación de los archivos de configuración residentes en éstas.
- Grupos de recursos, que servidores individuales pueden compartir con otros miembros de otros grupos, tales como manejo de transacciones; los grupos también definen la distribución de servidores a los manejadores de recursos en los cuales operan.
- Servidores que distribuyen a los procesos las peticiones del sistema; la lógica del negocio es implantada en estos servidores; la configuración permite a un administrador configurar múltiples servidores de uno o más grupos instalados en más de una maquina.
- Los servicios con los cuales el prototipo procesa; atributos a nivel del servicio que incluyen factores de carga, tiempo de proceso de un servicio y prioridad de un servicio con respecto a otros servicios.

Los tres primeros atributos de configuración mencionados anteriormente, definen los elementos de proceso (nodos de proceso), atributos globales y especificaciones correspondientes a los recursos maestros. Los grupos, servidores, y servicios se enfocan en los recursos de la aplicación distribuida: este prototipo define grupos de componentes de servidores que ofrecen servicios; un número de instancias de servidor pueden ser instalados en varias maquinas, y en el siguiente nivel puede manejar individualmente los servicios publicados y su prioridad.

El Proceso Administrador de los Recursos

El archivo de configuración corresponde con estructuras de datos en tiempo de ejecución. El proceso administrador de los recursos sirve como una base de información compartida derivada del archivo de configuración. Este proceso reside en cada nodo participante en el prototipo de acuerdo a la indicación del archivo de configuración. Este proceso sirve como la base de datos de servicios para la aplicación distribuida; provee información de la ubicación de los objetos de la aplicación y funciona como repositorio de las estadísticas de esta. Esta base de información provee al prototipo de los elementos necesarios para llevar a cabo distribución cliente/servidor de manera dinámica, también implantar funciones como balanceo de cargas de trabajo, seguridad, y coordinación de transacciones.

Manejador de Transacciones

Provee los servicios de la aplicación distribuida como son los siguientes: nombramiento, ruteo de mensajes, balanceo de cargas, manejo de la configuración, manejo de transacciones, y seguridad. Incluye también la estructura del proceso administrador de los recursos de la aplicación, y los servicio para mantener y actualizar esta información. Algunos atributos clave de este elemento son discutidos a continuación.

Nombramiento de Servicios/Transparencia de Ubicación

El proceso administrador de los recursos de la aplicación actúa como un servidor de nombramiento para una aplicación y es replicado en cada nodo participante. Para otorgar un acceso rápido, este servidor existe como una estructura en la memoria compartida. El Manejador de Transacciones utiliza la capacidad de nombramiento del proceso anterior, configuración e información de estadísticas para llevar a cabo el balanceo de cargas de las peticiones, rutear las peticiones de los clientes basadas en los contenidos de

los datos y otorgar prioridad a estas peticiones. El Manejador de Transacciones distribuye estas peticiones lógicas a instancias especificas de servidores dentro del ambiente de proceso.

Dentro del prototipo los usuarios, clientes, BD y servicios de la aplicación no estan "atados" unos con otros; al ocultar la localización de los diferentes modulos de la aplicación se simplificó el desarrollo de la distribución de la aplicación y se mejora las capacidades de administración – por ejemplo, uno de los recursos como lo es un servidor puede ser movido a una nueva computadora sin causar algún impacto a los clientes u otros servidores.

El prototipo oculta lo complejo de una red y los múltiples protocolos que podrían ser usados en la comunicación entre las máquinas involucradas en la distribución de la aplicación; al desarrollar la proramación de esta aplicación no nos tuvimos que preocupar de cómo se hacen los accesos a la red o de cómo se hace el manejo de los bytes en otras máquinas – sólo nos ocupamos de invocar servicios, publicar eventos o encolar/desencolar mensajes.

Los clientes del prototipo no conocen la estructura interna de un servicio, y la implantación de un servicio puede ser cambiada sin tener que alterar la programación de los clientes; incluso los clientes están programados con un lenguaje lenguaje diferente al del servidor.

Ruteo de Datos

El ruteo de datos es un mecanismo empleado por el prototipo, donde una petición a un servicio es conducida a un servidor especifico basándose en un valor contenido en un determinado campo del buffer de datos. Ejemplo, en una aplicación distribuida la información de los clientes puede estar segmentada en varias bases de datos, al definir criterios de ruteo en la configuración de la aplicación se consigue independencia en la codificación de los servicios, es decir, si algún día cambian los criterios de ruteo sólo se modifica la configuración, manteniendo el código intacto.

Una de los usos más comunes del ruteo de datos es cuando existe la partición horizontal de una Base de Daos. En términos de Bases de Datos Relacionales, el particionamiento horizontal se lleva a cabo cuando los renglones lógicos de una tabla son divididos a través de varias tablas de acuerdo con sus valores llave (pro ejemplo, una Base de Datos singular, cuyo campo llave es el número telefónico, es horizontalmente particionada a través de dos máquinas por el código de área. De este manera, la Base de Datos en la computadora 1 contiene registros sólo para el código de área 908, mientras que la Base de Datos en la computadora 2 contiene registro sólo para el código de área 201.

Balanceo de Cargas

Para asegurar el máximo desempeño del prototipo, el Manejador de Transacciones lleva a cabo de manera automática el balanceo de cargas. Utilizando factores de carga por servicio, entrega una petición particular al servidor que tenga menos carga de trabajo. El Manejador de Transacciones determina la carga en un servidor al sumar los factores de carga de las peticiones por procesar.

Múltiples Servidores, Una Sola Cola (MSSQ)

Este tipo de modelo es encontrado en la mayoría de los Bancos. Existe una fila general y varias cajas dando servicio, los clientes son atendidos tan pronto como una caja queda libre, el tiempo de espera depende de la rapidez con que van siendo atendidos los clientes en cualquiera de las cajas. Por lo cual, este prototipo adopta este esquema para dar una mayor rapidez en la atención a los clientes que hacen una petición a los servicios, dando así un mejor tiempo de respuesta en comparación a los sistemas que no estan basados en componentes.

Los servidores de esta aplicación que pertenecen a un MSSQ, estan configurados con su propia cola de respuestas (la cual no es compartida con ningún otro servidor). Estos servidores necesitan su propia cola de respuestas para que cuando requieran hacer peticiones a otros servidores, las respuestas sean

regresadas al servidor que hizo la petición y que éstas no sean desencoladas por otros servidores que no hicieron la petición.

Desde el punto de vista de los que desarrollamos este prototipo, ninguna de las colas es visible independientemente de la configuración que se tenga para los servidores.

Manejo de Prioridades

El manejo de prioridades para las peticiones es otra de las características ofrecidas por el Manejador de Transacciones, incluidas dentro de este prototipo. Algunas peticiones de servicio requieren tener una prioridad mayor que otras, un ejemplo dentro del funcionamiento del prototipo es la prioridad mayor que tiene un deposito en relación con un retiro o una consulta. El manejo de prioridades ocurre en la cola de mensajes de un servidor.

La asignación de prioridades va desde 1 (que es la menor) hasta 100 (que es la mayor) para los servicios ofrecidos dentro de esta aplicación. Cuando una petición es enviada, es marcada con la prioridad que maneja el servicio que es llamado. Cuando dos peticiones llegan a la cola del servidor, la petición que tiene una prioridad mayor es desencolada.

Ambiente de Ejecución Robusto

El Manejador de Transacciones incluye varias características para soportar la alta disponibilidad del prototipo, como son verificación de la disponibilidad de proceso, verificación de "time-outs", reinicio y procedimientos de recuperación de servidores. También lleva acabo el control del flujo de actividad de la aplicación.

Seguridad

El Manejador de Transacciones provee al prototipo de servicios de autenticación, autorización, y control de acceso a través de una interface de seguridad, esta interface es similar al sistema de seguridad de Kerberos, y permite la integración con sistemas de seguridad como el mismo Kerberos u otros.
Para la administración de esta aplicación se aplicó un nivel apropiado de seguridad. Por ejemplo, una aplicación puede ser configurada para que todos servidores tengan un acceso restringido a los recursos compartidos, tales como la memoria compartida y las colas de mensajes. Esta restricción en el acceso usa los permisos del sistema operativo para que sólo el administrador tenga acceso a estos recursos. Como resultado de esto, los servidores deben ejecutar con la identidad y permisos del administrador para poder realizar modificaciones a los recursos del sistema. En esta aplicación el administrador puede incrementar el nivel de seguridad de autenticación y autorización para protejer accesos contra aplicaciones y usuarios no autorizados. Por ejemplo, se especificó que los clientes deben proveer una contraseña para la aplicación y una del usuario cuando quieren accesarla.

Procesamiento Distribuido de las Transacciones

Esta capacidad del prototipo garantiza la integridad de los datos, accesados a través de varios sitios o administrados por diferentes productos de bases de datos. El Manejador de Transacciones coordina las transacciones distribuidas para habilitar operaciones múltiples contra bases de datos heterogéneas. Este control lo lleva a cabo utilizando transacciones globales y supervisando el protocolo de "two phase-commit".

El Manejador de Transacciones, permite al prototipo tener coordinación para la recuperación de las transacciones globales al ocurrir eventos de falla en cualquier nodo, fallas de red o agotamiento del tiempo de acceso a registros en la Base de Datos. El Manejador de Transacciones usa la interfaz XA (de X/OPEN) para la comunicación con diversas Bases de Datos. Esta interfaz ha sido aceptada por X/OPEN como la interfaz estándar para el control de transacciones distribuidas.

Debido a que el alto desempeño y el flujo de las trasacciones son sumamente críticas en un sistema OLTP en producción, como el presente prototipo, el software Manejador de Transacciones DTP usa algoritmos diseñados para minimizar la escritura a disco y tráfico en la red que pueda obstruir el flujo de las transacciones . Entre otros atributos, la implantación del Manejador de Transacciones DTP aprovecha técnicas conocidas como el cooordinador de migraciones y optimizaciones de "one-phase commit".

Administración

Una de las características más importantes de este prototipo es la separación entre las funciones del diseño de la aplicación y la administración de la misma. Al administrar esta aplicación sólo es necesario enfocarse en un correcta implantación de los módulos de la misma sin tener que preocuparse del contenido del código.

El administrador posee un conjunto de herramientas comprensibles para la simplificación del proceso de administración. El administrador se enfoca en las actividades tales como:

- Activación y desactivación de la aplicación.
- Observar el comportamiento de la aplicación (por ejemplo, obteniendo estadísticas).
- Detectar y responder a condiciones de fallas (por ejemplo, migrar servidores a otras máquinas).
- Detectar y responder a condiciones de peso (por ejemplo, inicializar nuevas instancias de servidores, suspender ciertos servicios, o cambiar atributos como valores de tiempos de respuesta).
- Detectar problemas de seguridad (por ejemplo, accesos no autorizados a servicios).

El prototipo soporta arquitecturas para la solución de problemas críticos, así como la administración de aplicaciones distribuidas. Las interfases administrativas del prototipo incluyen una interfaz comprensible de comandos en línea y una interfaz gráfica.

Definición de una aplicación centralizada

El Manejador de Transacciones permite al administrador del prototipo definir dentro de un archivo: el hardware, software, y los recursos de red. Es posible declarar dentro del archivo de configuración del prototipo cuando deben ser iniciados los servidores y servicios, así como cuando deben ser migrados en caso de ocurrir alguna falla en un procesador. Deben ser asignadas diversas características para los servidores de la aplicación, dentro de los cuales se incluyen esquemas de información, criterios de recuperación de procesos y períodos de "time-out".

El Manejador de Transacciones provee un manejo central para la configuración y herramientas para inicialización autómatica, terminación o administración de aplicaciones distribuidas.

Transparencia de la Administración

Los clientes y servidores de este prototipo desconocen la manera en que se lleva a cabo el balanceo de cargas de trabajo, cómo es aplicado el ruteo de los mensajes hacia los servidores, cómo se realiza la seguridad e incluso de cómo son agrupados los servicios dentro de los servidores; el administrador de esta aplicación puede cambiar libremente la configuración (por ejemplo, reubicar servidores o agregar máquinas nuevas) sin tener que alterar clientes o servidores; en general, existe una separación de lo concerniente a la administración con respecto a la lógica de la aplicación.

El subsistema de administrativo de este prototipo esta diseñado para proveer al administrador un control centralizado de la configuración de la aplicación, monitoreo, manejo por defecto, manejo de seguridad y manejo del desempeño de un dominio. Este subsistema, además, está diseñado para automatizar

aspectos críticos del manejo de la distribución de la aplicación. Por ejemplo, ejecuta levantamiento automático y recuperación de servidores, manejo del balanceo de cargas y el contexto del ruteo sensitivo al contenido de los datos, controla el tiempo de duración de un llamado o una transacción y puede de manera automática iniciar una transacción antes de que un servicio en particular sea llamado.

Desde la consola central, el administrador puede tomar decisiones basadas en una vista global de la aplicación. Puede observar el estatus de cada parte de la aplicación, cambiar dinámicamente la configuración, visualizar las estadísticas de la aplicación y monitorear los clientes que estan haciendo uso de la aplicación.

Reconfiguración Dinámica

Dentro de la funcionalidad del prototipo los servicios pueden ser dados de alta o dados de baja de manera dinámica, éstos también pueden hacerse disponibles de una forma selectiva. Elementos tales como: maquinas, grupos, servidores y servicios pueden ser agragados a la configuración sin dar de baja al sistema.

Se pueden asignar diversos parámetros a la configuración del prototipo como: el "time-out" en caso de que ocurra alguna falla. El Manejador de Transacciones permite a los servidores y servicios de un procesador ser migrados a otro procesador sin interrumpir la ejecución del prototipo.

Estándar TX de X/OPEN

El prototipo integra en su funcionalidad a la interfaz estándar TX de X/OPEN para la definición y manejo de las transacciones. Especificamente, la interfaz TX permite definir los límites de las transacciones dentro del prototipo, por lo tanto, las tareas que ejecutan los servicios pueden ser tratadas como una unidad atómica. Dentro de un Manejador de Transacciones, el trabajo ejecutado por varios servicios, accediendo a diversas Bases de Datos a través de muchas computadoras, es visto como una unidad

atómica de trabao en la realización de un "commit" o "roliback". Esto mantiene a todas la Bases de Datos sincronizadas, aunque ocurran fallas en las máquinas.

El prototipo hace uso de mecanismos optimizados para el manejo de transacciones distribuidas. Estos mecanismos explotan procesos asincronos para poder completar una transacción en servidores administrativos en vez de servidores aplicativos. Esta asincronia permite a los servidores aplicativos estar libres para procesar llamados a servicios en favor de diferentes transacciones, sin tener que esperar que se complete cualquier transacción, ésto se traduce en un mejor flujo para la aplicación.

Las completaciones de la transacciones son controladas por el servidor manejador de transacciones (TMS por sus siglas en inglés). El TMS es definido por el administrador cuando se especifica un grupo de servidores. Un grupo de servidores es asociado con un particular manejador de recursos transaccionales. Cuando el grupo de servidores es activado por el administrador, dos o más instancias de TMS son automaticamente inicializados. El TMS espera por la terminación de mensajes de transacciones que llegan a su cola de mensajes. Un singular TMS puede coordinar la terminacion de una transacción con otros TMSs. Este mecanismo de coordinación de transacciones usa protocolos optimizados para reducir los registros a disco y el número de mensajes requeridos para completar la transacción.

Finalmente, el TMS usa un archivo de bitácora, llamado bitacóra de transacciones o TLOG, para grabar información necesitada para la recuperación de una transacción por si ocurre alguna falla. Existe un archivo TLOG en cada máquina activa definida. El TLOG es creado por el administrador. Cuando un grupo de servidores es migrado de una máquina a otra, el TLOG debe ser descargado y vuelto a cargar en la nueva máquina.

Portabilidad

Este prototipo puede ejecutarse en numerosas plataformas y provee interoperabilidad entre ellas. Esto permitió que al iniciar el desarrollo nos concentraramos en la aplicación, en vez del manejo de la red y de la heterogeneidad del formato de los datos entre las diversas plataformas.

Este prototipo en su componente servidor, puede ejecutarse en una amplia variedad de plataformas apegadas a sistemas POSIX como UnixWare de SCO, IBM RS6000 AIX, HP – UX, y Digital Unix para sistemas POSIX como SCO's Open Desktop, así como otros tipos de plataformas como Windows NT, y Netware de Novell.

Capitulo 5

Este capitulo describe el proceso de desarrollo del Prototipo de Tarjeta de Crédito.

Desarrollo del Prototipo

Herramientas de Desarrollo.

Herramientas de Desarrollo Cliente.

La herramienta de desarrollo elegida para el Cliente de esta tesis es Visual Basic 5.0.

Algunas de sus características son:

- Esta basado en lenguaje BASIC de fácil aprendizaje.
- Esta diseñado para ser un lenguaje de programación para Windows, por lo que su principal orientación es escribir programas bajo el esquema de manejo de eventos.
- Posee un ambiente de desarrollo gráfico que permite crear programas funcionales agregando iconos en una pantalla si escribir una solo línea de código.

Se eligió Visual Basic porque permite al prototipo interactuar con el usuario a través de una Interfaz gráfica (GUI), que generalmente es visualmente más agradable y más fácil de usar.

Herramientas de Desarrollo Servidor.

Para la escritura de los Servidores Aplicativos de utilizo ANSI C usando como compilador a Microsoft Visual C++ 5.0

El lenguaje de programación C está caracterizado por ser de uso general, con una sintaxis sumamente compacta y de alta portabilidad.

Es un lenguaje de programación de alto nivel desarrollado en Bell Labs, que es capaz de manipular la computadora a bajo nivel, tal como lo haría un lenguaje ensamblador, esta particularidad, junto con el hecho de no poseer operaciones de entrada-salida, manejo de arreglo de caracteres, de asignación de memoria, etc. puede al principio parecer un grave defecto; sin embargo el hecho de que estas operaciones se realicen por medio de llamadas a funciones contenidas en librerías externas al lenguaje en sí, es el que confiere al mismo su alto grado de portabilidad, independizándolo del hardware sobre el cual corren los programas.

El lenguaje C puede ser compilado al lenguaje máquina de casi todas las computadoras. Por ejemplo, el UNIX está escrito en este lenguaje y se ejecuta en una amplia variedad de computadoras.

El lenguaje C se programa como una serie de funciones que se llaman unas a otras para el procesamiento. Aún el cuerpo del programa es una función llamada "main" (principal). Las funciones son muy flexibles, permitiendo a los programadores la elección entre el uso bibliotecas estándares que se provee con el compilador, el uso de funciones de terceros creadas por otros proveedores de C, o el desarrollo de sus propias funciones.

Comparado con otros lenguajes de programación de alto nivel, el C parece complicado. Su apariencia intrincada se debe a su extrema flexibilidad.

Visual C++ es la versión de Microsoft del lenguaje de programación C++, que a su vez esta basado en el lenguaje C. C++ es una mejora al lenguaje C que provee la funcionalidad de la programación orientada a objetos.

Herramienta de integración de sistemas.

La herramienta de integración de sistema elegido es BEA TUXEDO.

BEA TUXEDO constituye un sistema de middleware para desarrollar y ejecutar las aplicaciones cliente/servidor que son decisivas para las empresas. Además de controlar el procesamiento de transacciones distribuidas, se encarga de las funciones de mensajería y de toda la gama de servicios necesarios para crear y utilizar aplicaciones que engloban a toda la empresa. El sistema hace posible la creación de aplicaciones que abarcan una diversidad de plataformas de hardware, base de datos y sistemas operativos, con todos los privilegios para mezclar y combinar esas plataformas de la manera más adecuada para el entorno de la aplicación.

Es el núcleo del BEA Distributed Applications Framework (Marco de Aplicaciones Distribuidas de BEA), un grupo de middleware para crear y administrar aplicaciones cliente/servidor cuyo carácter es decisivo en el logro de un objetivo. Este marco comprende productos que facilitan la conectividad a través de múltiples entornos operativos, servicios de desarrollo y funciones administrativas.

Muchas aplicaciones diseñadas con BEA TUXEDO logran las siguientes ventajas:

- Rendimiento equivalente al de un mainframe, pero mucho más económico.
- Un aumento hasta del 400 por ciento en el rendimiento de la aplicación
- Independencia del sistema, de la red y de la base de datos.

Esta herramienta mejora el modelo básico de cliente/servidor de tal forma que permite la asignación flexible entre los clientes y servidores. En cuanto a la administración de las relaciones entre los clientes y los servidores, BEA TUXEDO aporta todas las cualidades propias de las aplicaciones que procesan transacciones en línea (OLTP):

- Capacidad para miles de usuarios concurrentes.
- Capacidad para grandes volúmenes de datos.
- Alto rendimiento en el procesamiento de transacciones.
- Entradas y salidas de la aplicación predecibles.
- Tiempo de respuestas cortos y predecibles.
- Alto grado de integridad y seguridad de los datos.
- Acceso concurrente a las bases de datos.
- Extensa disponibilidad de la aplicación, incluido el procesamiento 7x24.

La herramienta cumple con las normas de X/OPEN de The Open Group y funciona con más de 30 plataformas de hardware y sistemas operativos, entre ellos UNIX, NT y sistemas patentados. Es

totalmente compatible con las bases de datos que cumplen con XA, como por ejemplo CA/Ingress, DB2, Informix, ISAM-XA, Microsoft SQL Server, Oracle y Sybase.

La arquitectura de BEA TUXEDO se centra en su API de alto nivel, la ATMI (Interfaz del Administrador de Aplicación a Transacción o Application to Transaction Manager Interface), La ATMI, que consiste en 30 llamadas simples, ha sido adoptada por The Open Group como una API X/Open estándar. Por medio de ATMI, ofrece:

- Fácil construcción de clientes y servidores, se puede elegir entre más de 35 GLs y herramientas CASE para crear clientes y servidores BEA TUXEDO.
- Servidores de conversación para las aplicaciones de soporte de decisiones.
- Búferes tipificados que minimizan la codificación y decodificación de las aplicaciones y funciones de copia de memoria que mejoran el rendimiento de la aplicación.
- Encaminamiento dinámico dependiente de los datos.
- Jerarquización de las solicitudes de servicio según su importancia.

Con el fin de habilitar la alta disponibilidad de las aplicaciones, BEA TUXEDO proporciona:

- Ajuste dinámico de las aplicaciones, el cual facilita una respuesta instantánea a los cambios en las cargas y rendimiento efectivo del procesamiento de las aplicaciones.
- Administración centralizada de aplicaciones distribuidas.
- Equilibrio de cargas, reinicio del servidor y recuperación automáticas y programables.
- Sistema completo de registro de errores.
- Replicación de la capacidad del servidor para eliminar los puntos únicos de falla de la aplicación.

BEA TUXEDO proporciona integridad y seguridad absoluta de los datos en forma indetectable durante las actualizaciones, al mismo tiempo que mantiene la atomicidad del trabajo por medio de una asignación de dos etapas y características de autenticación y autorización.

Por medio de BEA TUXEDO los administradores pueden fácilmente controlar y administrar una aplicación, mejorar su rendimiento o reconfigurarlo durante su ejecución. Las interfaces administrativas comprenden una interfaz completa de línea de comandos e instrucciones, una interfaz programática y una base para administrar información que sirve para ejecutar BEA TUXEDO como una aplicación administrada dentro de un marco administrativo mucho más grande.

BEA TUXEDO proporciona autenticación y autorización de servicios por medio de una interfaz de seguridad, que se basa en Kerberos.

Incorpora un sistema de eventos de transacciones basados en el paradigma de publicación y suscripción. Este método de comunicación es particularmente útil en una aplicación en la cual es necesario notificar sobre un evento a un usuario o programa para poder tomar una decisión.

BEA TUXEDO esta basado en diferentes componentes:
Administrador de transacciones.
Estación de trabajo.
Servicios de cola de espera.
Dominios.

El Administrador de transacciones proporciona los servicios indispensables de las aplicaciones distribuidas: nombramiento, encaminamiento de mensajes, equilibrio de cargas, administración de configuraciones, administración de transacciones y seguridad.

El componente Estación de trabajo ofrece soporte completo para SO Mac, OS/2, UNIX, Windows 3.1/95 y NT esto permite que las aplicaciones tengan clientes remotos, sin requerir que resida en cada máquina la infraestructura completa.

El componente Servicios de cola de espera proporciona un marco para mensajes que sirve para crear aplicaciones empresariales distribuidas. Como alternativa a la comunicación en línea basándose en solicitudes y respuestas, el empleo de colas de espera permite que las aplicaciones empresariales se comuniquen por medio de colas de almacenamiento estables de una manera asíncrona o sujeta al tiempo. Las colas habilitan el trabajo en curso y el flujo de trabajo de las aplicaciones, la entrega y finalización garantizada de transacciones, la entrega garantizada de solicitudes urgentes, el control de transacciones para las solicitudes de cola y la flexibilidad por medio de la fácil duplicación de los servicios y la información.

Los dominios permiten configurar los servidores BEA TUXEDO en grupos administrativamente autónomos, denominados dominios. Este marco es lo que permite la ejecución de aplicaciones de gran escala y alta rendimiento a través de centenares o millares de nodos. También proporciona un alto nivel de flexibilidad administrativa y seguridad.

Manejador de Base de Datos.

El manejador de Base de Datos utilizado en el Prototipo es Microsoft SQLServer 6.5.
Microsoft SQLServer 6.5 es una Manejador de Base de Datos Relacionales diseñado específicamente para aplicaciones cliente/servidor.

La manera en que los Servidores Aplicativos acceden a la Base de Datos es por medio de SQL Inmerso.
El SQL (Structured Query language) o lenguaje de consulta estructurado es un estándar ampliamente aceptado por la industria para la definición y manipulación de datos y la protección, acceso y control de la información. SQL es un concepto originado por las Base de datos Relacionales y sustenta su filosofía en el uso de tablas, índices, llaves, renglones y columnas para identificar locaciones de almacenamiento.

El SQL inmerso permite colocar declaraciones SQL dentro de un programa creado en un lenguaje tradicional. Se pueden delimitar las declaraciones SQL dentro de estos programas con parámetros de comienzo y finalización definidos por el lenguaje propietario.

Cuando se compila un programa que contiene SQL Inmerso, se utiliza un precompilador para las declaraciones de SQL. El precompilador reemplaza las definiciones SQL con un equivalente de código fuente del lenguaje propietario. Después de precompilar el programa se utiliza el compilador del lenguaje propietario para compilar el código fuente resultante.

En el prototipo el lenguaje tradicional es C y el precompilador es SQLC de Microsoft.

Funcionalidad del prototipo.

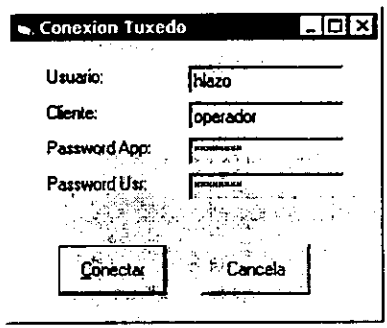
El objetivo del prototipo es realizar las principales funciones de un sistema encargado de administrar Tarjetas de Crédito.

El prototipo está diseñado para ser utilizado por un operador de la compañía emisora de tarjetas de crédito (no por el propietario de la tarjeta de crédito), las funciones que puede realizar el operador son:

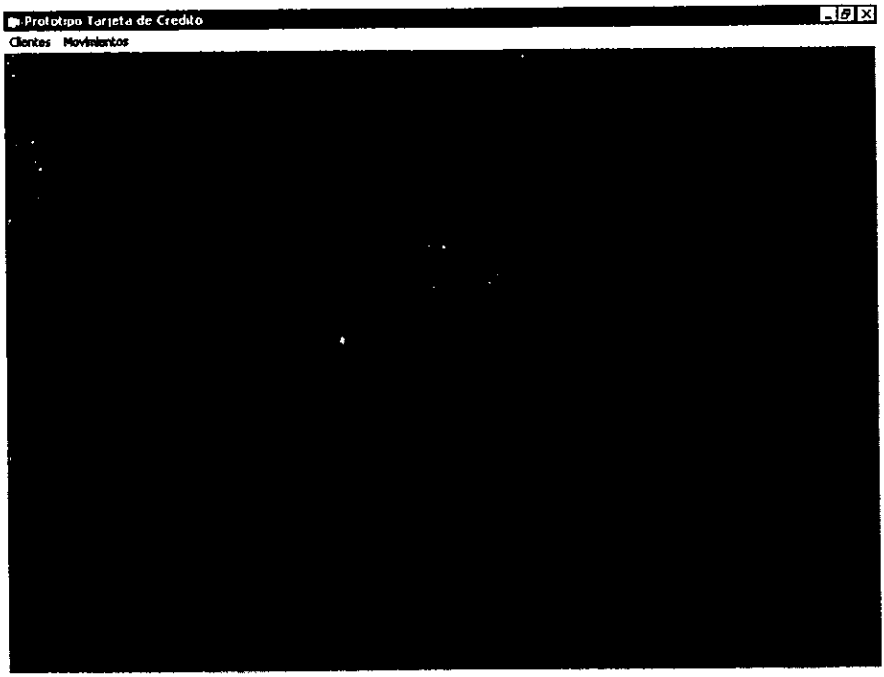
Captura de datos de nuevos clientes.
Consulta de datos de los clientes.
Cargo a cuentas.
Abonos a cuentas.
Consulta de saldos y movimientos.

La aplicación se creo bajo el concepto de Interface de Documento Múltiple (MDI, multiple document interface), por lo que todas las operaciones se encuentran bajo un mismo Menú de Opciones.

La pantalla inicial utiliza un servicio de autentificación para comprobar que el usuario exista y tenga facultades para utilizar la aplicación.



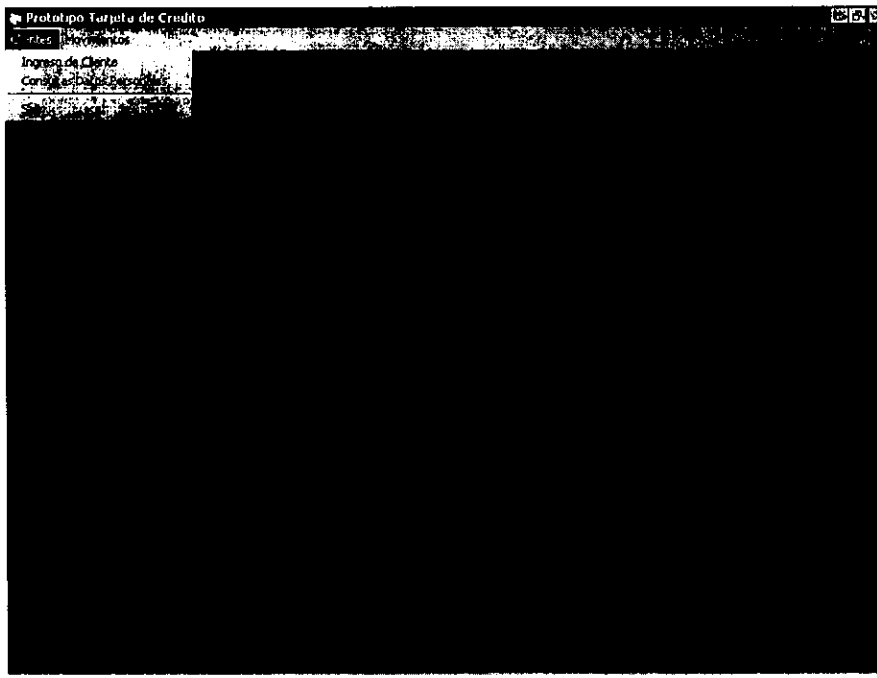
La pantalla principal tiene las siguientes opciones:



La opción Clientes

Clientes se divide en las siguientes opciones:

- Consulta de datos personales.
- Ingreso de Cliente.
- Salida.



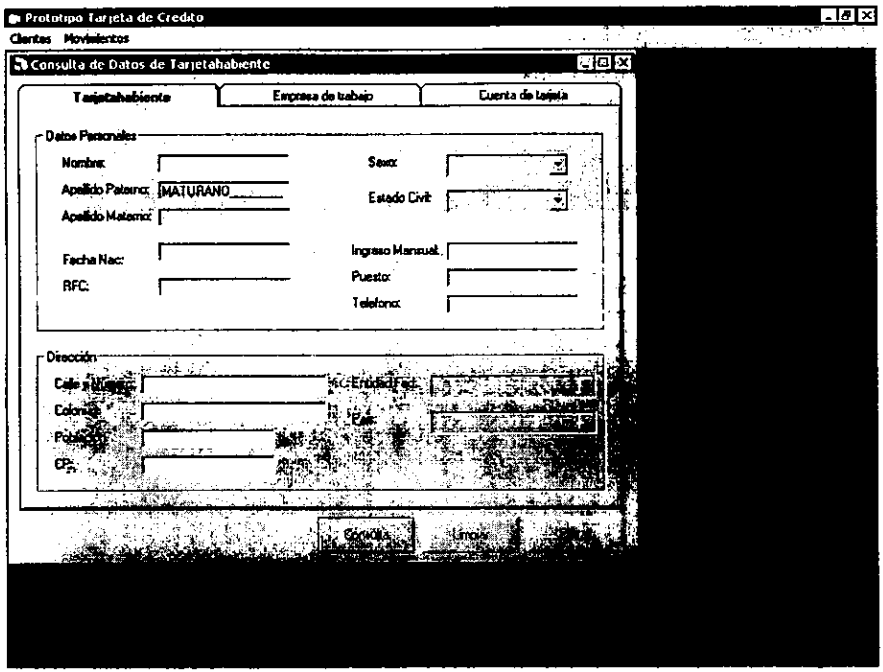
Dentro de la opción Consulta de Datos Personales se desplegaran los datos de los Clientes divididos en tres rubros:

Datos del tarjetahabiente.

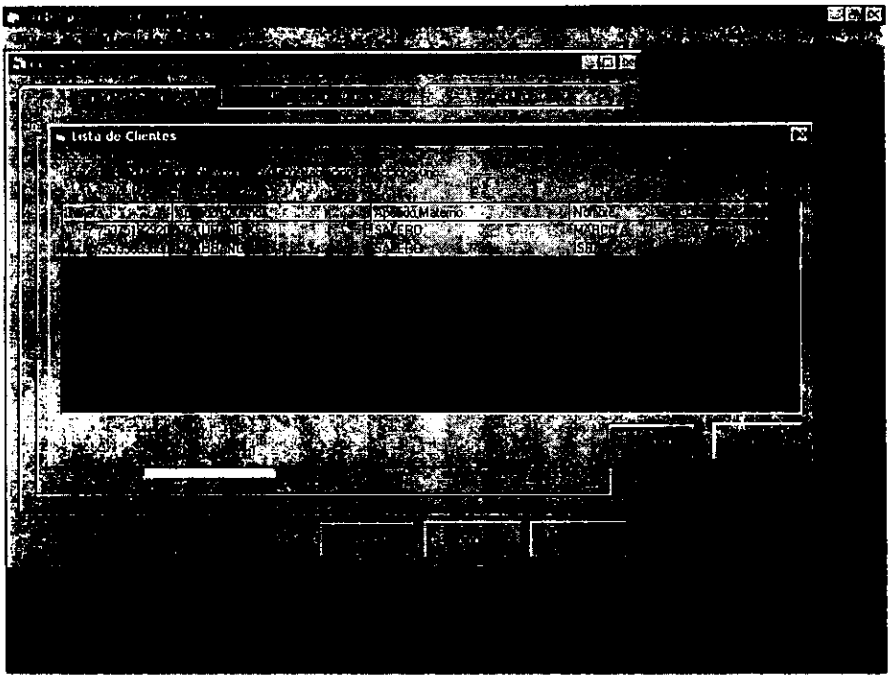
Datos de la empresa donde labora el tarjetahabiente.

Datos de la cuenta.

El operador podrá elaborara consultas a partir del Nombre (Apellido Paterno y Apellido Materno) o mediante el Número de la tarjeta.

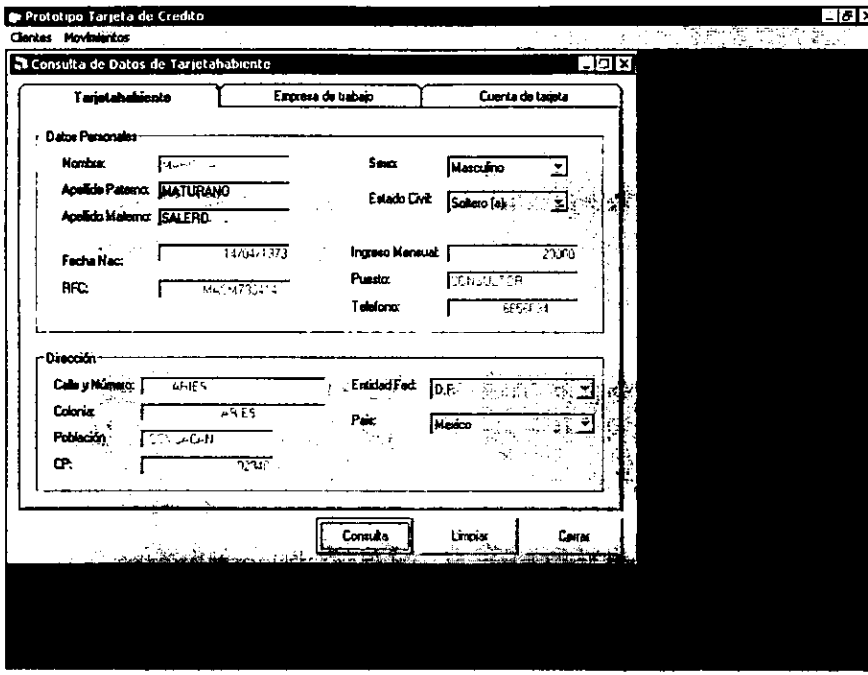


Si más de un registro cumple con los criterios establecidos se muestra una pantalla donde se debe eligir el registro del cual se desea mostrar el detalle.

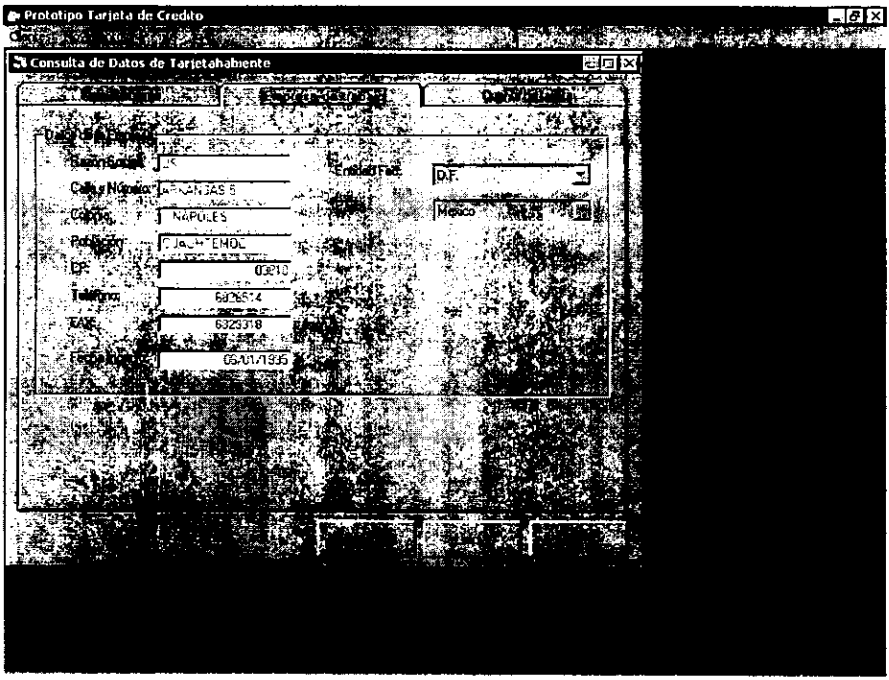


El detalle se despliega en las siguientes pantallas:

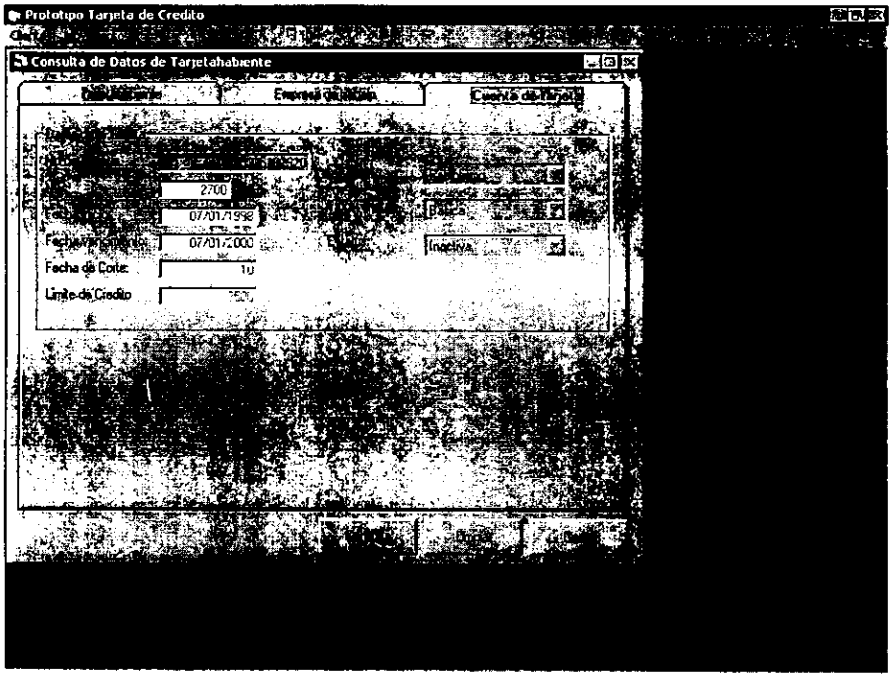
Datos del tarjetahabiente:



Datos de la empresa donde labora el tarjetahabiente:



Datos de la cuenta:



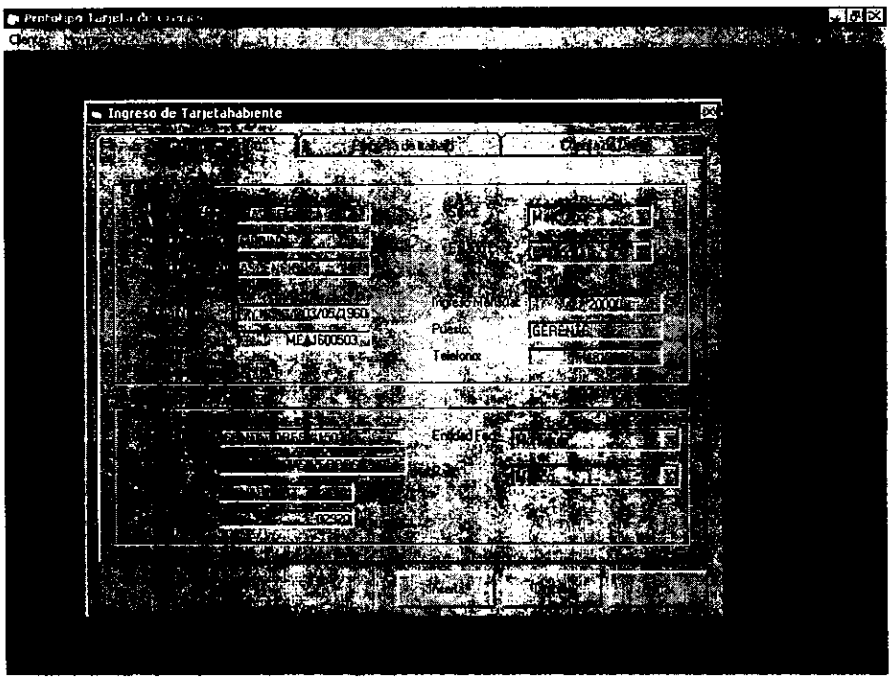
En la opción Ingreso de Cliente el operador deberá capturar la información sobre el posible cliente, el sistema se encargara de validar si el solicitante cumple con todos los requisitos para convertirse en cliente, las validaciones que se realizan son:

- Que el salario del solicitante sea mayor a 5,000 pesos.
- Que tenga una antigüedad en su empleo mayor a 2 años.

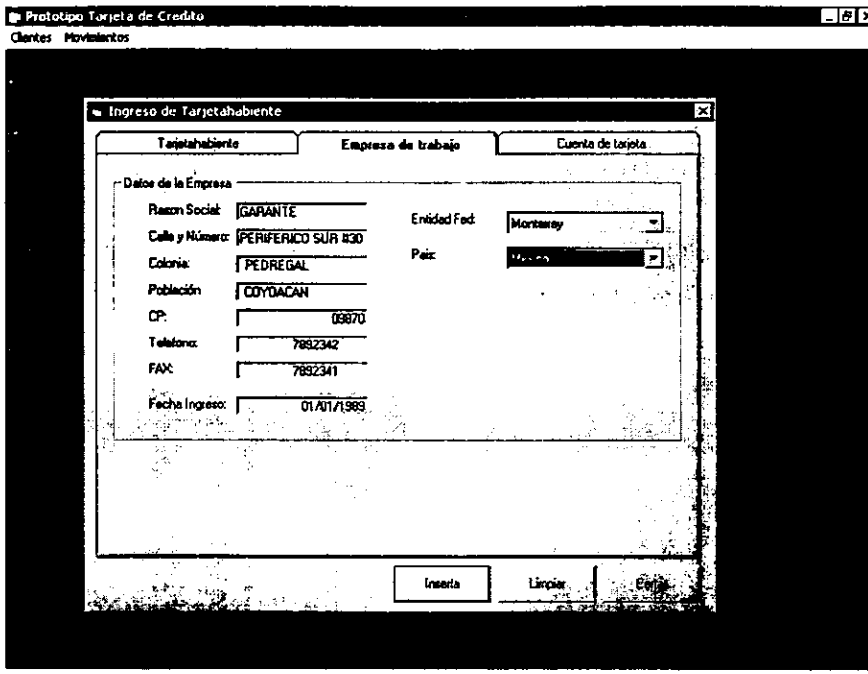
Si el solicitante es rechazado como cliente se le informara al operador y sus datos no se almacenara, si el solicitante es aceptado se generara su Número de Tarjeta y se ingresaran sus datos, notificando al operador.

La información es capturada en las siguientes pantallas.

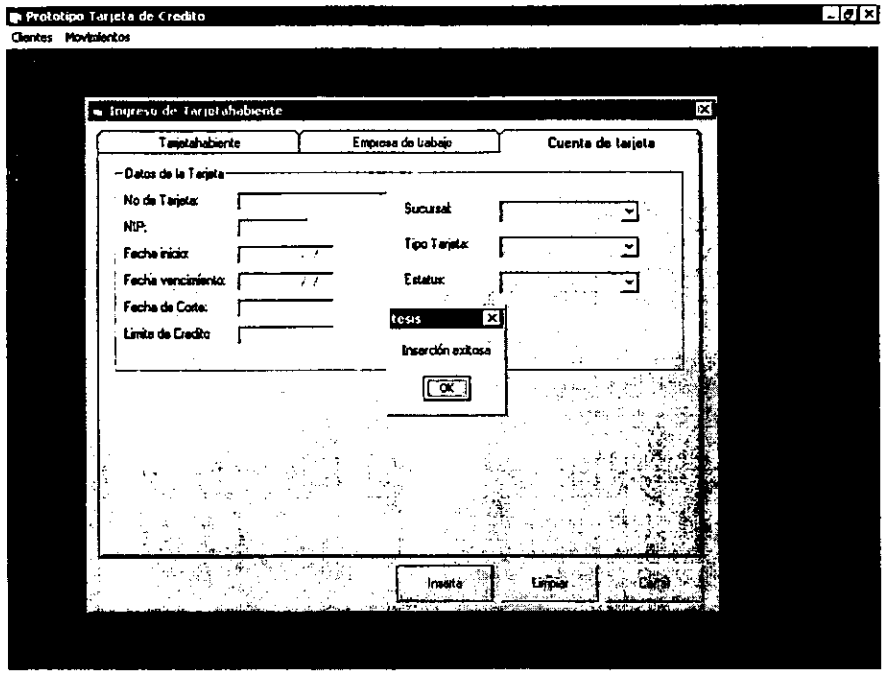
Pantalla de captura de datos generales.

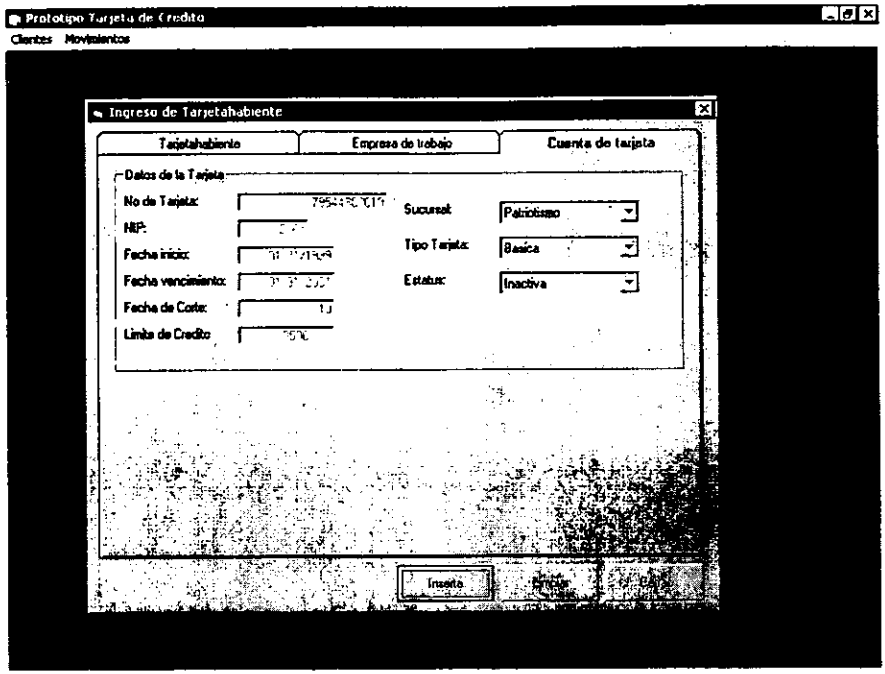


Pantalla de captura de datos del Patrón.



Los datos de la tarjeta se asignan cuando la transacción es exitosa.

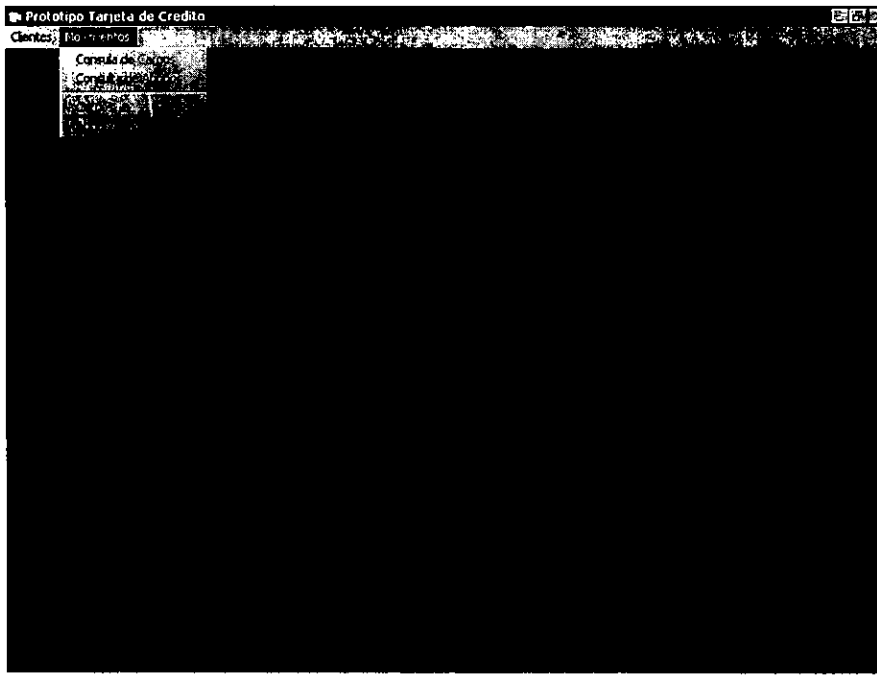



Prototipo Tarjeta de Credito
Clientes Movimientos
Ingreso de Tarjetahabiente
Tarjetahabiente
Empresa de trabajo
Cuenta de tarjeta
Datos de la Tarjeta
No de Tarjeta:
NIP:
Fecha inicio:
Fecha vencimiento:
Fecha de Corte:
Limite de Credito:
Sucursal:
Patriotismo
Tipo Tarjeta:
Basica
Estatus:
Inactiva
Inserta

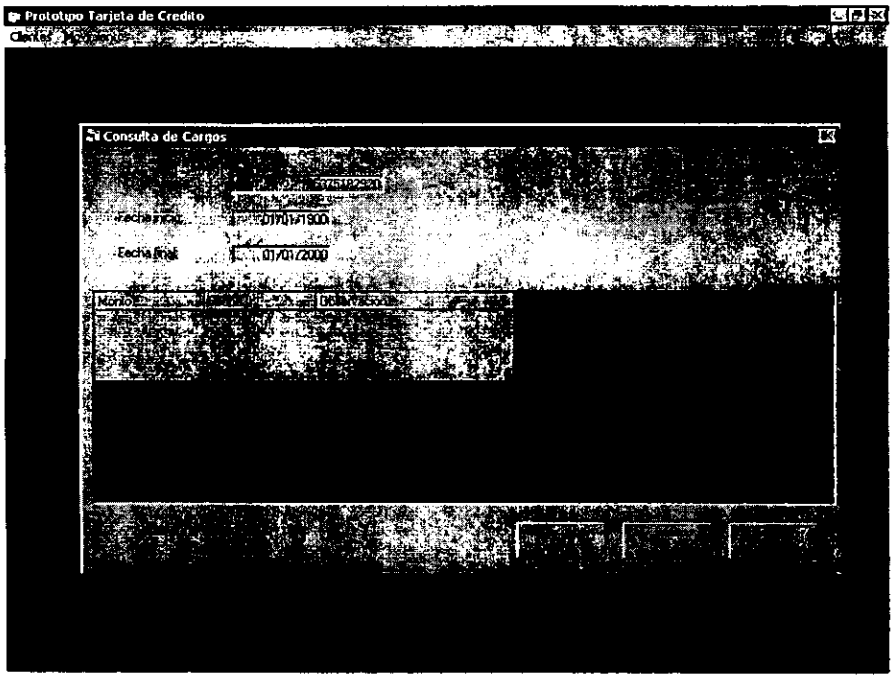
La opción Movimientos

La opción Movimientos se divide en las opciones:

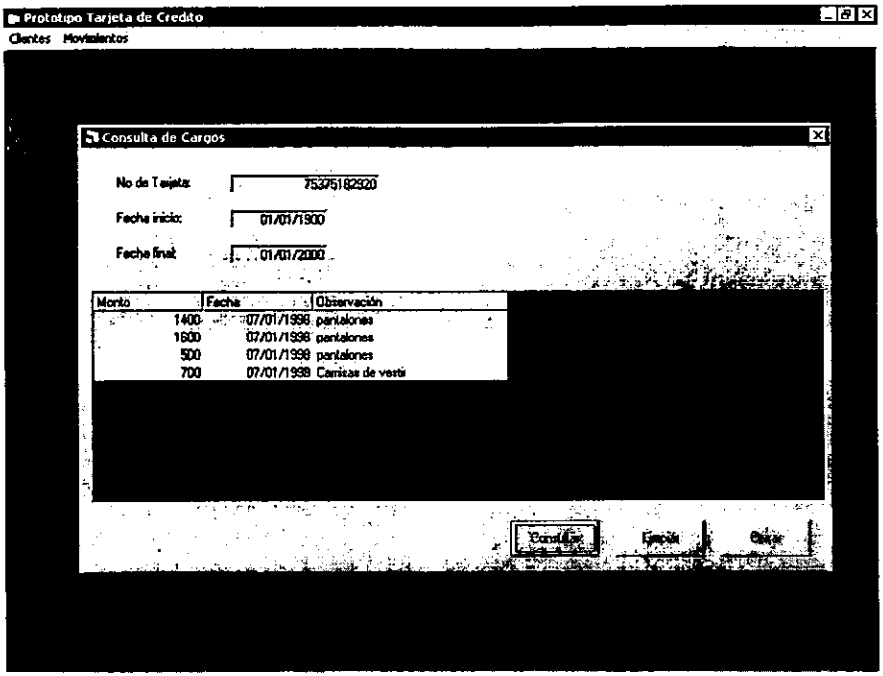
- Consulta de Cargos.
- Consulta de Abonos.
- Cargos.
- Abonos.



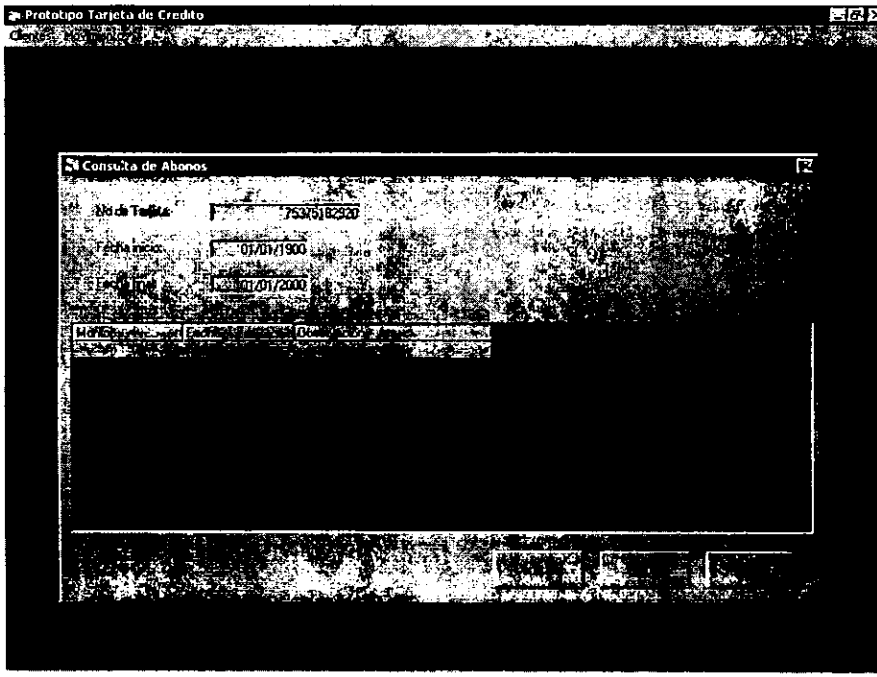
En la Consulta de Cargos se debe ingresar el Número de Tarjeta y el Periodo del que se desea mostrar.



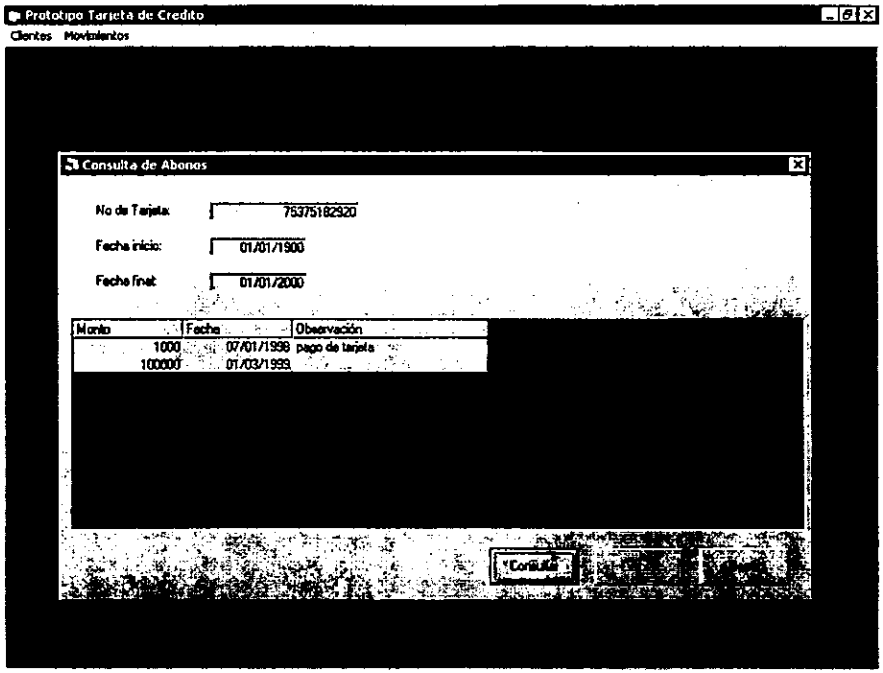
Como resultado el sistema mostrara los Cargos correspondientes a la Tarjeta en el periodo establecido.



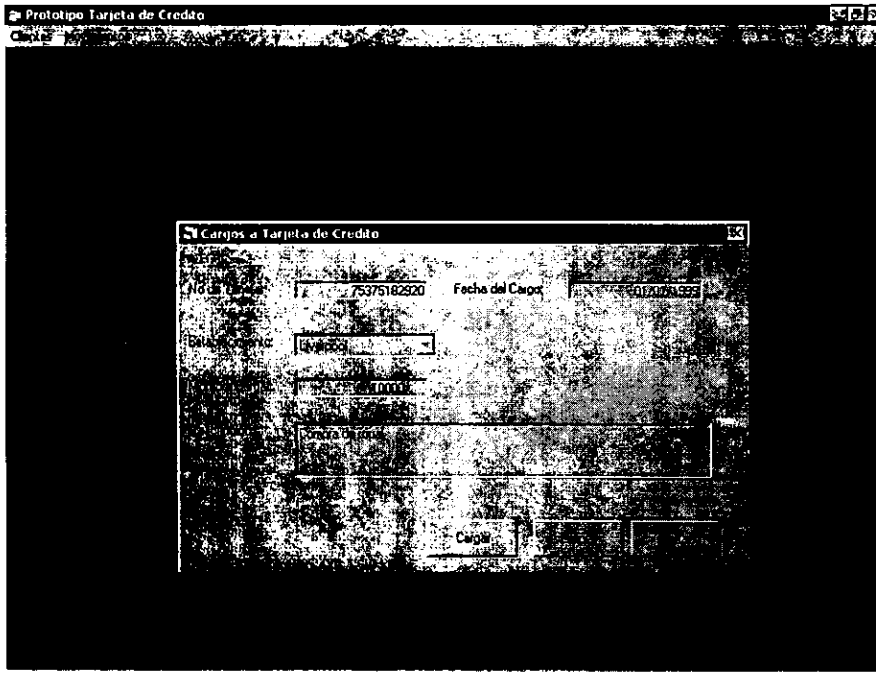
En la Consulta de Abonos se debe ingresar el Número de Tarjeta y el Periodo del que se desea mostrar.



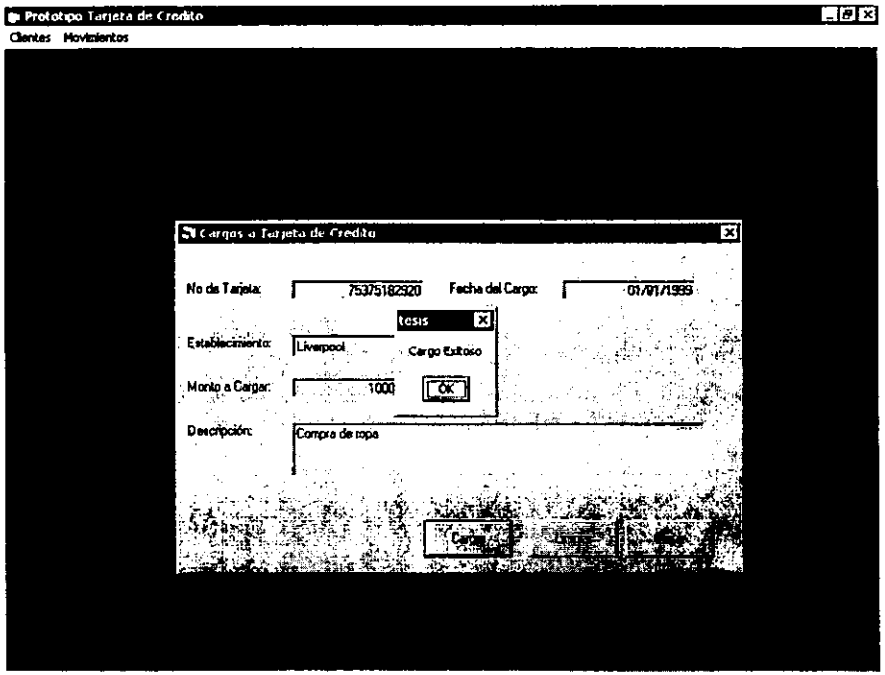
Como resultado el sistema mostrara los Abonos correspondientes a la Tarjeta en el periodo establecido.



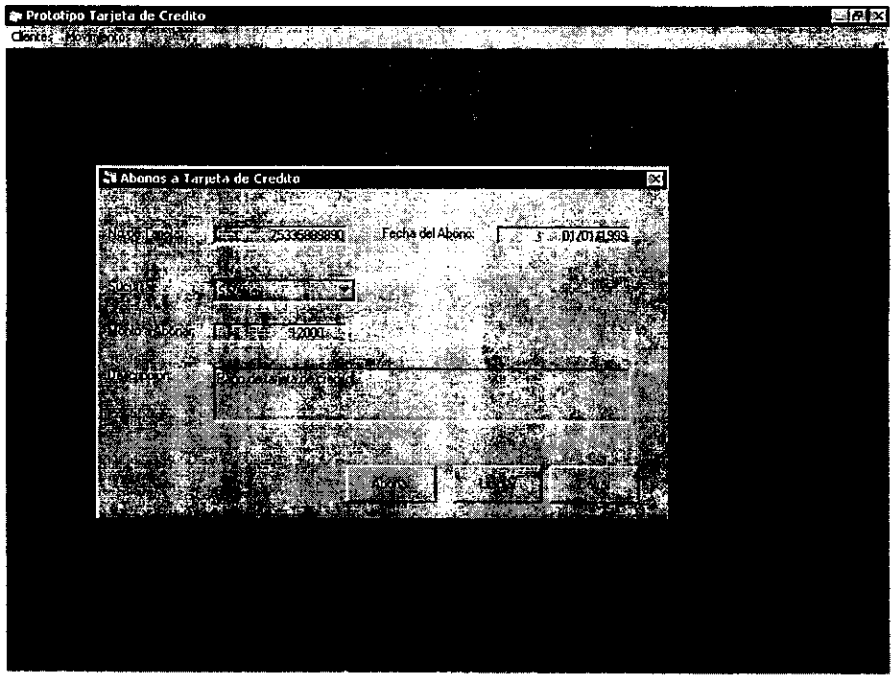
En Cargos el usuario debera capturar el Número de Tarjeta al que desea hacerle el cargo, el monto del cargo, la fecha del cargo, el establecimiento donde se realiza el cargo y opcionalmente una descripción.



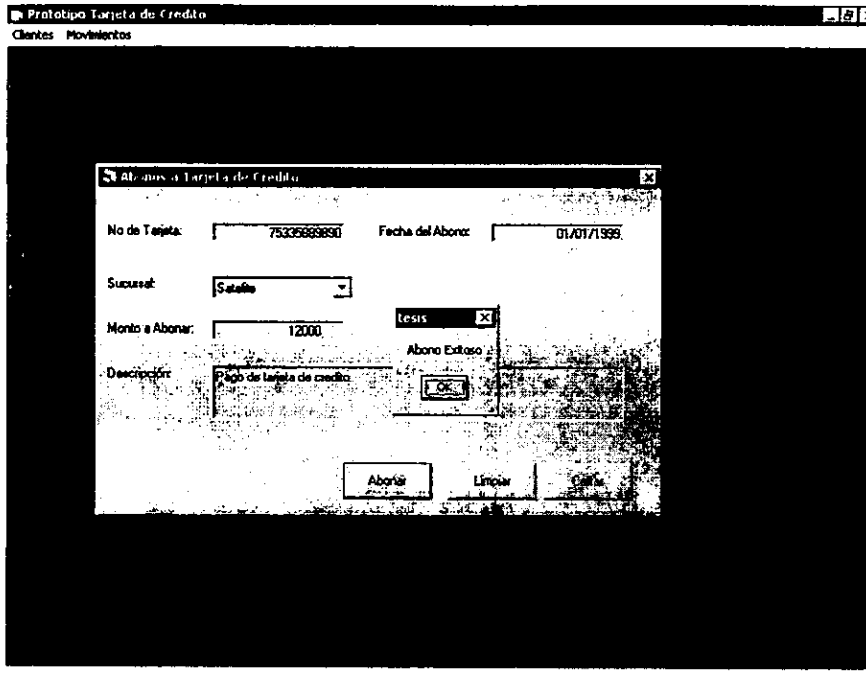
Como respuesta el prototipo informa si el cargo se realizó exitosamente.



En Abonos el usuario debera capturar el Número de Tarjeta al que desea hacerle el abono, el monto del abono, la fecha del abonos, la sucursal donde se realiza el abono y opcionalmente una descripción.



Como respuesta el prototipo informa si el abono se realizó exitosamente.



Desarrollo del Prototipo

Cliente

Los clientes utilizados para este prototipo son clientes remotos porque residen como entidades individuales enlazados con el servidor por medio de la red.

Para poder interactuar con los Servidores Aplicativos el Cliente necesita un conjunto de archivos adicionales que forman parte de las librerías BEA TUXEDO /WS (Estación de Trabajo), que permiten utilizar el ATMI dentro de Visual Basic.

Antes de iniciar la comunicación con los Servidores Aplicativos los Clientes deben conocer la ubicación del WSL con el que establecerán la comunicación, esta ubicación se establece mediante una variable de ambiente o por el uso de una instrucción del ATMI

tuxputenv ("WSNADDR=//wookie:2000")

La conexión a la Herramienta de Integración (BEA TUXEDO) involucra varios procesos:
1. Reservar memoria para el Buffer de intercambio de información, este proceso se realiza por medio de la instrucción del ATMI *tpalloc*.

initptr = tpalloc("TPINIT", "", 0)

TPINIT es una estructura de datos que contiene el nombre del usuario que desea iniciar la conexión, el grupo al que pertenece y sus Passwords.

2. Iniciar la conexión por medio de *tpinit*:

rc = tpinit(ByVal initptr)

3. Se recomienda que se libere le memoria reservada para el Buffer de intercambio de información a través de *tpfree*.

Después de que el Cliente se ha conectado puede iniciar la interacción con los Servidores Aplicativos mediante el llamado a Servicios, este proceso se da en los siguientes pasos:

1. Reservar memoria para el Buffer de intercambio de información por medio de la instrucción del ATMI *tpalloc*.
2. Invocar y enviar el Buffer al Servicio por medio de *tpcall*.
3. Finalmente liberar la memoria reservada mediante *tpfree*.

Generalmente la información que se envía a los Servicios es resultado de una entrada del operador, y también generalmente después de un llamado a un Servicio se despliega el resultado de este llamado al operador.

Al terminar las operaciones que requería el Cliente debe desconectarse de BEA TUXEDO, esta desconexión se realiza mediante la instrucción *tpterm*:

rc = tpterm()

A continuación se lista el Código Fuente del Cliente, solo se muestra el código relacionado con la construcción del cliente remoto.

Forma de Conexión:

```
Private Sub Cmd_coneccion_Click()

    Dim tinit As Long
    Dim X As String
    Dim usrname As String
    Dim cltname As String
    Dim passwd As String
    Dim grpname As String
    Dim usrpasswd As String
    Dim flags As Long

    usrname = String(32, " ")
    cltname = String(32, " ")
    passwd = String(32, " ")
    grpname = String(32, " ")
    usrpasswd = String(32, " ")

    usrname = TxtUsr.Text
    cltname = TxtClt.Text
    passwd = TxtPwd.Text
    grpname = ""
    usrpasswd = TxtUsrPwd.Text
    flags = 0

    'Aloja bufer de conexion
    initptr = tpalloc("TPINIT", "", 0)
    If initptr = 0 Then
        MsgBox "Error alojando buffer TPINIT", , "ERROR"
        End
    End If

    tinit = initptr
    rc = lstrcpy(ByVal tinit&, ByVal usrname$) ' Nombre de usuario
    tinit = tinit + 32
    rc = lstrcpy(ByVal tinit&, ByVal cltname$) ' Nombre de cliente
    tinit = tinit + 32
    rc = lstrcpy(ByVal tinit&, ByVal passwd$)  'password aplicativo
    tinit = tinit + 32
    rc = lstrcpy(ByVal tinit&, ByVal grpname$) 'grupo
    tinit = tinit + 32
    X$ = "0"
    rc = lstrcpy(ByVal tinit&, ByVal X$)  'Banderas: 0
    tinit = tinit + 4
    X$ = Str(Len(usrpasswd$))
    rc = lstrcpy(ByVal tinit&, ByVal X$) 'Datalen, longitud de password
    tinit = tinit + 4
    rc = lstrcpy(ByVal tinit&, ByVal usrpasswd$)  ' data, passwd de usuario
    tinit = tinit + 4

    ' Realizamos la conexion con la aplicacion con el apuntador original
    rc = tpinit(ByVal initptr)
    If rc = -1 Then
        tpfree (initptr)
        TuxError ("Error Tuxedo")
        MsgBox "Error en conexión"
    Else
        tpfree (initptr)
        Unload Me
        Menu.Show
    End If

End Sub
```

Forma Menú, Extracción de Catálogos:

```
Private Sub MDIForm_Load()

Dim Ocurrencias, i, posicion As Long

'Lee los catálogos del Sistema

buffer.ptr = tpalloc("FML32", "", 8192)
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

rc = tpcall("SVC_CATALOGO", buffer.ptr&, 0, buffer.ptr&, lon, 0)
If rc < 0 Then
    TuxError "Error en catalogos"
End If

Ocurrencias = Foccur32(buffer.ptr, CAT_ID)

For i = 0 To Ocurrencias - 1
    lon = Flen32(buffer.ptr, CAT_ID, i)
    rc = Fget32(buffer.ptr, CAT_ID, i, V_CAT_ID, lon)

    lon = Flen32(buffer.ptr, DES_ID, i)
    rc = Fget32(buffer.ptr, DES_ID, i, ByVal V_DES_ID, lon)

    lon = Flen32(buffer.ptr, DES_DES, i)
    rc = Fget32(buffer.ptr, DES_DES, i, ByVal V_DES_DES, lon)

    posicion = 0
    Select Case V_CAT_ID
    Case CAT_ID_SEXO
        On Error Resume Next
        posicion = UBound(CAT_SEXO, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_SEXO(1 To 2, 1 To posicion)
        CAT_SEXO(1, posicion) = V_DES_ID
        CAT_SEXO(2, posicion) = V_DES_DES
    Case CAT_ID_ESTADO_CIVIL
        On Error Resume Next
        posicion = UBound(CAT_ESTADO_CIVIL, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_ESTADO_CIVIL(1 To 2, 1 To posicion)
        CAT_ESTADO_CIVIL(1, posicion) = V_DES_ID
        CAT_ESTADO_CIVIL(2, posicion) = V_DES_DES
    Case CAT_ID_TIPO_TARJETA
        On Error Resume Next
        posicion = UBound(CAT_TIPO_TARJETA, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_TIPO_TARJETA(1 To 2, 1 To posicion)
        CAT_TIPO_TARJETA(1, posicion) = V_DES_ID
        CAT_TIPO_TARJETA(2, posicion) = V_DES_DES
    Case CAT_ID_ESTATUS_TARJETA
        On Error Resume Next
        posicion = UBound(CAT_ESTATUS_TARJETA, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_ESTATUS_TARJETA(1 To 2, 1 To posicion)
        CAT_ESTATUS_TARJETA(1, posicion) = V_DES_ID
        CAT_ESTATUS_TARJETA(2, posicion) = V_DES_DES
    Case CAT_ID_SUCURSAL
        On Error Resume Next
        posicion = UBound(CAT_SUCURSAL, 2)
```

```
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_SUCURSAL(1 To 2, 1 To posicion)
        CAT_SUCURSAL(1, posicion) = V_DES_ID
        CAT_SUCURSAL(2, posicion) = V_DES_DES
    Case CAT_ID_ESTABLECEMIENTO
        On Error Resume Next
        posicion = UBound(CAT_ESTABLECEMIENTO, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_ESTABLECEMIENTO(1 To 2, 1 To posicion)
        CAT_ESTABLECEMIENTO(1, posicion) = V_DES_ID
        CAT_ESTABLECEMIENTO(2, posicion) = V_DES_DES
    Case CAT_ID_ENTIDAD_FEDERATIVA
        On Error Resume Next
        posicion = UBound(CAT_ENTIDAD_FEDERATIVA, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_ENTIDAD_FEDERATIVA(1 To 2, 1 To posicion)
        CAT_ENTIDAD_FEDERATIVA(1, posicion) = V_DES_ID
        CAT_ENTIDAD_FEDERATIVA(2, posicion) = V_DES_DES
    Case CAT_ID_PAIS
        On Error Resume Next
        posicion = UBound(CAT_PAIS, 2)
        On Error GoTo 0
        posicion = posicion + 1
        ReDim Preserve CAT_PAIS(1 To 2, 1 To posicion)
        CAT_PAIS(1, posicion) = V_DES_ID
        CAT_PAIS(2, posicion) = V_DES_DES
    End Select

Next

tpfree (buffer.ptr)

End Sub
```

Forma Menú Terminación de la Aplicación:

```
Private Sub M_Salir_Click()

    rc = tpterm()
    End
End Sub
```

Forma de Ingreso de Clientes:

```
Private Sub Valida_Click()

Dim Error_aplicacion As Integer

If Not Valida_datos Then
    Exit Sub
End If

buffer.ptr = tpalloc("FML32", "", 8192)
```

```
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

rc = Fchg32(buffer.ptr, CLI_APE_PAT, 0, ByVal V_CLI_APE_PAT, 0)
rc = Fchg32(buffer.ptr, CLI_APE_MAT, 0, ByVal V_CLI_APE_MAT, 0)
rc = Fchg32(buffer.ptr, CLI_NOM, 0, ByVal V_CLI_NOM, 0)
rc = Fchg32(buffer.ptr, CLI_RFC, 0, ByVal V_CLI_RFC, 0)
rc = Fchg32(buffer.ptr, CLI_DIR, 0, ByVal V_CLI_DIR, 0)
rc = Fchg32(buffer.ptr, CLI_COL, 0, ByVal V_CLI_COL, 0)
rc = Fchg32(buffer.ptr, CLI_POB, 0, ByVal V_CLI_POB, 0)
rc = Fchg32(buffer.ptr, CLI_ENT_FED, 0, ByVal V_CLI_ENT_FED, 0)
rc = Fchg32(buffer.ptr, CLI_PAIS, 0, ByVal V_CLI_PAIS, 0)
rc = Fchg32(buffer.ptr, CLI_CP, 0, ByVal V_CLI_CP, 0)
rc = Fchg32(buffer.ptr, CLI_TEL, 0, ByVal V_CLI_TEL, 0)
rc = Fchg32(buffer.ptr, CLI_CIVIL, 0, ByVal V_CLI_CIVIL, 0)

rc = Fchg32(buffer.ptr, CLI_FEC_NAC, 0, ByVal V_CLI_FEC_NAC, 0)
rc = Fchg32(buffer.ptr, CLI_ING_MENS, 0, V_CLI_ING_MENS, 0)
rc = Fchg32(buffer.ptr, CLI_SEXO, 0, ByVal V_CLI_SEXO, 0)
rc = Fchg32(buffer.ptr, CLI_PUESTO, 0, ByVal V_CLI_PUESTO, 0)
rc = Fchg32(buffer.ptr, PAT_NOM, 0, ByVal V_PAT_NOM, 0)
rc = Fchg32(buffer.ptr, PAT_DIR, 0, ByVal V_PAT_DIR, 0)
rc = Fchg32(buffer.ptr, PAT_COL, 0, ByVal V_PAT_COL, 0)
rc = Fchg32(buffer.ptr, PAT_POB, 0, ByVal V_PAT_POB, 0)
rc = Fchg32(buffer.ptr, PAT_ENT_FED, 0, ByVal V_PAT_ENT_FED, 0)
rc = Fchg32(buffer.ptr, PAT_PAIS, 0, ByVal V_PAT_PAIS, 0)
rc = Fchg32(buffer.ptr, PAT_CP, 0, ByVal V_PAT_CP, 0)
rc = Fchg32(buffer.ptr, PAT_TEL, 0, ByVal V_PAT_TEL, 0)
rc = Fchg32(buffer.ptr, PAT_FAX, 0, ByVal V_PAT_FAX, 0)
rc = Fchg32(buffer.ptr, PAT_INI_LAB, 0, ByVal V_PAT_INI_LAB, 0)

'Llamado al servicio

rc = tpcall("SVC_INSERTA", buffer.ptr&, 0, buffer.ptr&, lon, 0)
If rc < 0 Then
    Error_aplicacion = gettpurcode()
    Select Case Error_aplicacion
    Case 3
        MsgBox "Cliente ya existe en la Base de Datos"
    Case 10
        MsgBox "Cliente no cumple todos los requisistos."
    Case Else
        TuxError ("Error en la Inserción.")
    End Select
    Exit Sub
End If

MsgBox "Inserción exitosa"

'Extrae datos del Buffer de Regreso

lon = Flen32(buffer.ptr, TAR_SUC, 0)
rc = Fget32(buffer.ptr, TAR_SUC, 0, ByVal V_TAR_SUC, lon)

lon = Flen32(buffer.ptr, TAR_TIPO, 0)
rc = Fget32(buffer.ptr, TAR_TIPO, 0, ByVal V_TAR_TIPO, lon)

lon = Flen32(buffer.ptr, TAR_STATUS, 0)
rc = Fget32(buffer.ptr, TAR_STATUS, 0, ByVal V_TAR_STATUS, lon)

lon = Flen32(buffer.ptr, TAR_NUMERO, 0)
rc = Fget32(buffer.ptr, TAR_NUMERO, 0, ByVal V_TAR_NUMERO, lon)

lon = Flen32(buffer.ptr, TAR_NIP, 0)
rc = Fget32(buffer.ptr, TAR_NIP, 0, ByVal V_TAR_NIP, lon)

lon = Flen32(buffer.ptr, TAR_FEC_INI, 0)
rc = Fget32(buffer.ptr, TAR_FEC_INI, 0, ByVal V_TAR_FEC_INI, lon)
```

```
lon = Flen32(buffer.ptr, TAR_FEC_VEN, 0)
rc = Fget32(buffer.ptr, TAR_FEC_VEN, 0, ByVal V_TAR_FEC_VEN, lon)

lon = Flen32(buffer.ptr, TAR_FEC_CORTE, 0)
rc = Fget32(buffer.ptr, TAR_FEC_CORTE, 0, ByVal V_TAR_FEC_CORTE, lon)

lon = Flen32(buffer.ptr, TAR_LIMI_CRED, 0)
rc = Fget32(buffer.ptr, TAR_LIMI_CRED, 0, V_TAR_LIMI_CRED, lon)

'Despliega los Datos

Tab_datos_tarjetahabiente.Tab = 2

cb_tar_suc.Text = Busca_descripcion(V_TAR_SUC, CAT_SUCURSAL)
cb_tar_tipo.Text = Busca_descripcion(V_TAR_TIPO, CAT_TIPO_TARJETA)
cb_tar_status.Text = Busca_descripcion(V_TAR_STATUS, CAT_ESTATUS_TARJETA)
mk_tar_numero.Text = V_TAR_NUMERO
mk_tar_nip.Text = V_TAR_NIP
mk_tar_fec_ini.Text = V_TAR_FEC_INI
mk_tar_fec_ven.Text = V_TAR_FEC_VEN
mk_tar_fec_corte.Text = V_TAR_FEC_CORTE
mk_tar_limi_cred.Text = V_TAR_LIMI_CRED


'Liberera Buffer
tpfree (buffer.ptr)

End Sub
```

Forma de Consulta de datos generales del Tarjetahabiente:

```
Private Sub Cmd_consulta_Click()

Dim Error_aplicacion As Integer
Dim Ocurrencias As Long
Dim i As Long

If Not Valida_datos Then
    Exit Sub
End If

buffer.ptr = tpalloc("FML32", "", 8192)
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

'V_CLI_APE_PAT
If Trim(V_CLI_APE_PAT) <> "" Then
    rc = Fchg32(buffer.ptr, CLI_APE_PAT, 0, ByVal V_CLI_APE_PAT, 0)
End If

'V_CLI_APE_MAT
If Trim(V_CLI_APE_MAT) <> "" Then
    rc = Fchg32(buffer.ptr, CLI_APE_MAT, 0, ByVal V_CLI_APE_MAT, 0)
End If

'V_TAR_NUMERO
If Trim(V_TAR_NUMERO) <> "" Then
    rc = Fchg32(buffer.ptr, TAR_NUMERO, 0, ByVal V_TAR_NUMERO, 0)
End If

'Llamado al servicio
```

```
rc = tpcall("SVC_CONSULTA", buffer.ptr&, 0, buffer.ptr&, lon, 0)
If rc < 0 Then
    Error_aplicacion = gettpurcode()
    Select Case Error_aplicacion
    Case 5
        MsgBox "No existe clientes con el criterio especificado."

    Case Else
        TuxError ("Error en la Consulta.")
    End Select
    Exit Sub
End If

'Verifica el número de Ocurrencias

Ocurrencias = Foccur32(buffer.ptr, CLI_APE_PAT)
If Ocurrencias > 1 Then
    ReDim A_datos_clientes(1 To 5, 1 To Ocurrencias)
    For Ocurrencia = 0 To (Ocurrencias - 1)
        A_datos_clientes(1, Ocurrencia + 1) = Str(Ocurrencia)

        lon = Flen32(buffer.ptr, TAR_NUMERO, Ocurrencia)
        rc = Fget32(buffer.ptr, TAR_NUMERO, Ocurrencia, ByVal V_TAR_NUMERO, lon)
        A_datos_clientes(2, Ocurrencia + 1) = V_TAR_NUMERO

        lon = Flen32(buffer.ptr, CLI_APE_PAT, Ocurrencia)
        rc = Fget32(buffer.ptr, CLI_APE_PAT, Ocurrencia, ByVal V_CLI_APE_PAT, lon)
        A_datos_clientes(3, Ocurrencia + 1) = V_CLI_APE_PAT

        lon = Flen32(buffer.ptr, CLI_APE_MAT, Ocurrencia)
        rc = Fget32(buffer.ptr, CLI_APE_MAT, Ocurrencia, ByVal V_CLI_APE_MAT, lon)
        A_datos_clientes(4, Ocurrencia + 1) = V_CLI_APE_MAT

        lon = Flen32(buffer.ptr, CLI_NOM, Ocurrencia)
        rc = Fget32(buffer.ptr, CLI_NOM, Ocurrencia, ByVal V_CLI_NOM, lon)
        A_datos_clientes(5, Ocurrencia + 1) = V_CLI_NOM
    Next
        Ocurrencia = -1
        ListaClientes.Show vbModal
        'No se selecciono ningun elemento del Grid
        If Ocurrencia = -1 Then
            tpfree (buffer.ptr)
            Exit Sub
        End If
    Else
        Ocurrencia = 0
End If


'Extrae datos del Buffer de Regreso

'Datos del Cliente

lon = Flen32(buffer.ptr, CLI_SEXO, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_SEXO, Ocurrencia, ByVal V_CLI_SEXO, lon)

lon = Flen32(buffer.ptr, CLI_CIVIL, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_CIVIL, Ocurrencia, ByVal V_CLI_CIVIL, lon)

lon = Flen32(buffer.ptr, CLI_ENT_FED, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_ENT_FED, Ocurrencia, ByVal V_CLI_ENT_FED, lon)

lon = Flen32(buffer.ptr, CLI_PAIS, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_PAIS, Ocurrencia, ByVal V_CLI_PAIS, lon)

lon = Flen32(buffer.ptr, CLI_NOM, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_NOM, Ocurrencia, ByVal V_CLI_NOM, lon)

lon = Flen32(buffer.ptr, CLI_APE_PAT, Ocurrencia)
```

```
rc = Fget32(buffer.ptr, CLI_APE_PAT, Ocurrencia, ByVal V_CLI_APE_PAT, lon)

lon = Flen32(buffer.ptr, CLI_APE_MAT, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_APE_MAT, Ocurrencia, ByVal V_CLI_APE_MAT, lon)

lon = Flen32(buffer.ptr, CLI_FEC_NAC, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_FEC_NAC, Ocurrencia, ByVal V_CLI_FEC_NAC, lon)

lon = Flen32(buffer.ptr, CLI_RFC, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_RFC, Ocurrencia, ByVal V_CLI_RFC, lon)

lon = Flen32(buffer.ptr, CLI_DIR, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_DIR, Ocurrencia, ByVal V_CLI_DIR, lon)

lon = Flen32(buffer.ptr, CLI_COL, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_COL, Ocurrencia, ByVal V_CLI_COL, lon)

lon = Flen32(buffer.ptr, CLI_POB, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_POB, Ocurrencia, ByVal V_CLI_POB, lon)

lon = Flen32(buffer.ptr, CLI_CP, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_CP, Ocurrencia, ByVal V_CLI_CP, lon)

lon = Flen32(buffer.ptr, CLI_ING_MENS, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_ING_MENS, Ocurrencia, V_CLI_ING_MENS, lon)

lon = Flen32(buffer.ptr, CLI_PUESTO, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_PUESTO, Ocurrencia, ByVal V_CLI_PUESTO, lon)

lon = Flen32(buffer.ptr, CLI_TEL, Ocurrencia)
rc = Fget32(buffer.ptr, CLI_TEL, Ocurrencia, ByVal V_CLI_TEL, lon)

'Datos del patron

lon = Flen32(buffer.ptr, PAT_ENT_FED, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_ENT_FED, Ocurrencia, ByVal V_PAT_ENT_FED, lon)

lon = Flen32(buffer.ptr, PAT_PAIS, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_PAIS, Ocurrencia, ByVal V_PAT_PAIS, lon)

lon = Flen32(buffer.ptr, PAT_NOM, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_NOM, Ocurrencia, ByVal V_PAT_NOM, lon)

lon = Flen32(buffer.ptr, PAT_DIR, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_DIR, Ocurrencia, ByVal V_PAT_DIR, lon)

lon = Flen32(buffer.ptr, PAT_COL, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_COL, Ocurrencia, ByVal V_PAT_COL, lon)

lon = Flen32(buffer.ptr, PAT_POB, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_POB, Ocurrencia, ByVal V_PAT_POB, lon)

lon = Flen32(buffer.ptr, PAT_CP, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_CP, Ocurrencia, ByVal V_PAT_CP, lon)

lon = Flen32(buffer.ptr, PAT_TEL, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_TEL, Ocurrencia, ByVal V_PAT_TEL, lon)

lon = Flen32(buffer.ptr, PAT_FAX, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_FAX, Ocurrencia, ByVal V_PAT_FAX, lon)

lon = Flen32(buffer.ptr, PAT_INI_LAB, Ocurrencia)
rc = Fget32(buffer.ptr, PAT_INI_LAB, Ocurrencia, ByVal V_PAT_INI_LAB, lon)

'Datos de la tarjeta

lon = Flen32(buffer.ptr, TAR_SUC, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_SUC, Ocurrencia, ByVal V_TAR_SUC, lon)

lon = Flen32(buffer.ptr, TAR_TIPO, Ocurrencia)
```

```
rc = Fget32(buffer.ptr, TAR_TIPO, Ocurrencia, ByVal V_TAR_TIPO, lon)

lon = Flen32(buffer.ptr, TAR_STATUS, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_STATUS, Ocurrencia, ByVal V_TAR_STATUS, lon)

lon = Flen32(buffer.ptr, TAR_NUMERO, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_NUMERO, Ocurrencia, ByVal V_TAR_NUMERO, lon)

lon = Flen32(buffer.ptr, TAR_NIP, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_NIP, Ocurrencia, ByVal V_TAR_NIP, lon)

lon = Flen32(buffer.ptr, TAR_FEC_INI, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_FEC_INI, Ocurrencia, ByVal V_TAR_FEC_INI, lon)

lon = Flen32(buffer.ptr, TAR_FEC_VEN, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_FEC_VEN, Ocurrencia, ByVal V_TAR_FEC_VEN, lon)

lon = Flen32(buffer.ptr, TAR_FEC_CORTE, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_FEC_CORTE, Ocurrencia, ByVal V_TAR_FEC_CORTE, lon)

lon = Flen32(buffer.ptr, TAR_LIMI_CRED, Ocurrencia)
rc = Fget32(buffer.ptr, TAR_LIMI_CRED, Ocurrencia, V_TAR_LIMI_CRED, lon)

'Despliega los Datos
Tab_datos_tarjetahabiente.Tab = 0

'Datos del Cliente

'cb_cli_sexo
cb_cli_sexo.Text = Busca_descripcion(V_CLI_SEXO, CAT_SEXO)
'cb_cli_civil
cb_cli_civil.Text = Busca_descripcion(V_CLI_CIVIL, CAT_ESTADO_CIVIL)
'cb_cli_ent_fed
cb_cli_ent_fed.Text = Busca_descripcion(V_CLI_ENT_FED, CAT_ENTIDAD_FEDERATIVA)
'cb_cli_pais
cb_cli_pais.Text = Busca_descripcion(V_CLI_PAIS, CAT_PAIS)
'mk_cli_nom
mk_cli_nom.Text = V_CLI_NOM
'mk_cli_ape_pat
mk_cli_ape_pat.Text = V_CLI_APE_PAT
'mk_cli_ape_mat
mk_cli_ape_mat.Text = V_CLI_APE_MAT
'mk_cli_fec_nac
mk_cli_fec_nac.Text = Format(V_CLI_FEC_NAC, "dd/mm/yyyy")
'mk_cli_rfc
mk_cli_rfc.Text = V_CLI_RFC
'mk_cli_dir
mk_cli_dir.Text = V_CLI_DIR
'mk_cli_col
mk_cli_col.Text = V_CLI_COL
'mk_cli_pob
mk_cli_pob.Text = V_CLI_POB
'mk_cli_cp
mk_cli_cp.Text = V_CLI_CP
'mk_cli_ing_mens
mk_cli_ing_mens.Text = V_CLI_ING_MENS
'mk_cli_puesto
mk_cli_puesto.Text = V_CLI_PUESTO
'mk_cli_tel
mk_cli_tel.Text = V_CLI_TEL

'Datos del Patron

'cb_pat_ent_fed
cb_pat_ent_fed.Text = Busca_descripcion(V_PAT_ENT_FED, CAT_ENTIDAD_FEDERATIVA)
'cb_pat_pais
cb_pat_pais.Text = Busca_descripcion(V_PAT_PAIS, CAT_PAIS)
'mk_pat_nom
mk_pat_nom.Text = V_PAT_NOM
'mk_pat_dir
```

```
mk_pat_dir.Text = V_PAT_DIR
'mk_pat_col
mk_pat_col.Text = V_PAT_COL
'mk_pat_pob
mk_pat_pob.Text = V_PAT_POB
'mk_pat_cp
mk_pat_cp.Text = V_PAT_CP
'mk_pat_tel
mk_pat_tel.Text = V_PAT_TEL
'mk_pat_fax
mk_pat_fax.Text = V_PAT_FAX
'mk_pat_ini_lab
mk_pat_ini_lab.Text = V_PAT_INI_LAB

'Datos de la Tarjeta

'cb_tar_suc
cb_tar_suc.Text = Busca_descripcion(V_TAR_SUC, CAT_SUCURSAL)
'cb_tar_tipo
cb_tar_tipo.Text = Busca_descripcion(V_TAR_TIPO, CAT_TIPO_TARJETA)
'cb_tar_status
cb_tar_status.Text = Busca_descripcion(V_TAR_STATUS, CAT_ESTATUS_TARJETA)
'mk_tar_numero
mk_tar_numero.Text = V_TAR_NUMERO
'mk_tar_nip
mk_tar_nip.Text = V_TAR_NIP
'mk_tar_fec_ini
mk_tar_fec_ini.Text = V_TAR_FEC_INI
'mk_tar_fec_ven
mk_tar_fec_ven.Text = V_TAR_FEC_VEN
'mk_tar_fec_corte
mk_tar_fec_corte.Text = V_TAR_FEC_CORTE
'mk_tar_limi_cred
mk_tar_limi_cred.Text = V_TAR_LIMI_CRED

tpfree (buffer.ptr)


End Sub
```

Forma de Cargos:

```
Private Sub Cmd_cargo_Click()

Dim Error_aplicacion As Integer

If Not Valida_datos Then
    Exit Sub
End If

buffer.ptr = tpalloc("FML32", "", 8192)
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

rc = Fchg32(buffer.ptr, TAR_NUMERO, 0, ByVal V_TAR_NUMERO, 0)
rc = Fchg32(buffer.ptr, CAR_MONTO, 0, V_CAR_MONTO, 0)
rc = Fchg32(buffer.ptr, CAR_FEC, 0, ByVal V_CAR_FEC, 0)
rc = Fchg32(buffer.ptr, CAR_ESTABLE, 0, ByVal V_CAR_ESTABLE, 0)
rc = Fchg32(buffer.ptr, CAR_OBSER, 0, ByVal V_CAR_OBSER, 0)


rc = tpcall("SVC_CARGO", buffer.ptr&, 0, buffer.ptr&, lon, 0)
```

```
If rc < 0 Then
    Error_aplicacion = gettpurcode()
    Select Case Error_aplicacion
    Case 1
        MsgBox "Número de Tarjeta no existe."
    Case 2
        MsgBox "Crédito insuficiente."
    Case Else
        TuxError ("Error en el Abono.")
    End Select
    Exit Sub
End If

MsgBox "Cargo Exitoso"

'Liberera Buffer
tpfree (buffer.ptr)

End Sub
```

Forma de Abonos:

```
Private Sub Cmd_abono_Click()

Dim Error_aplicacion As Integer

If Not Valida_datos Then
    Exit Sub
End If

buffer.ptr = tpalloc("FML32", "", 8192)
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

rc = Fchg32(buffer.ptr, TAR_NUMERO, 0, ByVal V_TAR_NUMERO, 0)
rc = Fchg32(buffer.ptr, ABO_MONTO, 0, V_ABO_MONTO, 0)
rc = Fchg32(buffer.ptr, ABO_FEC, 0, ByVal V_ABO_FEC, 0)
rc = Fchg32(buffer.ptr, ABO_SUC, 0, ByVal V_ABO_SUC, 0)
rc = Fchg32(buffer.ptr, ABO_OBSER, 0, ByVal V_ABO_OBSER, 0)


rc = tpcall("SVC_ABONO", buffer.ptr&, 0, buffer.ptr&, lon, 0)
If rc < 0 Then
    Error_aplicacion = gettpurcode()
    Select Case Error_aplicacion
    Case 1
        MsgBox "Número de Tarjeta no existe."

    Case Else
        TuxError ("Error en el Abono.")
    End Select
    Exit Sub
End If

MsgBox "Abono Exitoso"

'Liberera Buffer
tpfree (buffer.ptr)

End Sub
```

Forma Consulta de Cargos:

```
Private Sub Cmd_consultar_Click()

'Iniciar Grid
Gd_cargos.Clear

Gd_cargos.Row = 0

Gd_cargos.Col = 0
Gd_cargos.Text = "Monto"
Gd_cargos.ColWidth(0) = 1500

Gd_cargos.Col = 1
Gd_cargos.Text = "Fecha"
Gd_cargos.ColWidth(1) = 1500

Gd_cargos.Col = 2
Gd_cargos.Text = "Observación"
Gd_cargos.ColWidth(2) = 2700

Dim Error_aplicacion As Integer
Dim Ocurrencias As Long

If Not Valida_datos Then
    Exit Sub
End If

buffer.ptr = tpalloc("FML32", "", 8192)
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

rc = Fchg32(buffer.ptr, CAR_FEC, 0, ByVal V_CAR_FEC_1, 0)
rc = Fchg32(buffer.ptr, CAR_FEC, 1, ByVal V_CAR_FEC_2, 0)
rc = Fchg32(buffer.ptr, TAR_NUMERO, 0, ByVal V_TAR_NUMERO, 0)

'Llamando al servicio

rc = tpcall("SVC_MOVTOS_CAR", buffer.ptr&, 0, buffer.ptr&, lon, 0)
If rc < 0 Then
    Error_aplicacion = gettpurcode()
    Select Case Error_aplicacion
    Case 1
        MsgBox "No existe el número de Tarjeta."
    Case Else
        TuxError ("Error en la Consulta.")
    End Select
    Exit Sub
End If

'Verifica el número de Ocurrencias


Ocurrencias = Foccur32(buffer.ptr, CAR_MONTO)

If Ocurrencias = 0 Then
    MsgBox "No existen cargos."
    tpfree (buffer.ptr)
    Exit Sub
End If

Gd_cargos.Rows = Ocurrencias + 1
```

```
    For Ocurrencia = 0 To (Ocurrencias - 1)
        lon = Flen32(buffer.ptr, CAR_MONTO, Ocurrencia)
        rc = Fget32(buffer.ptr, CAR_MONTO, Ocurrencia, V_CAR_MONTO, lon)
        'Imprime el valor
        Gd_cargos.Row = Ocurrencia + 1
        Gd_cargos.Col = 0
        Gd_cargos.Text = V_CAR_MONTO

        lon = Flen32(buffer.ptr, CAR_FEC, Ocurrencia)
        rc = Fget32(buffer.ptr, CAR_FEC, Ocurrencia, ByVal V_CAR_FEC, lon)
        'Imprime el valor
        Gd_cargos.Row = Ocurrencia + 1
        Gd_cargos.Col = 1
        Gd_cargos.Text = V_CAR_FEC

        lon = Flen32(buffer.ptr, CAR_OBSER, Ocurrencia)
        rc = Fget32(buffer.ptr, CAR_OBSER, Ocurrencia, ByVal V_CAR_OBSER, lon)
       'Imprime el valor
        Gd_cargos.Row = Ocurrencia + 1
        Gd_cargos.Col = 2
        Gd_cargos.Text = V_CAR_OBSER
Next

tpfree (buffer.ptr)

End Sub
```

Forma Consulta de Abonos:

```
Private Sub Cmd_consultar_Click()

'Iniciar Grid
Gd_abonos.Clear

Gd_abonos.Row = 0

Gd_abonos.Col = 0
Gd_abonos.Text = "Monto"
Gd_abonos.ColWidth(0) = 1500

Gd_abonos.Col = 1
Gd_abonos.Text = "Fecha"
Gd_abonos.ColWidth(1) = 1500

Gd_abonos.Col = 2
Gd_abonos.Text = "Observación"
Gd_abonos.ColWidth(2) = 2700

Dim Error_aplicacion As Integer
Dim Ocurrencias As Long

If Not Valida_datos Then
    Exit Sub
End If

buffer.ptr = tpalloc("FML32", "", 8192)
If Fielded32(buffer.ptr) <> 1 Then
    Fml32Error "Error alojando buffer"
End If

rc = Fchg32(buffer.ptr, ABO_FEC, 0, ByVal V_ABO_FEC_1, 0)
rc = Fchg32(buffer.ptr, ABO_FEC, 1, ByVal V_ABO_FEC_2, 0)
```

```
rc = Fchg32(buffer.ptr, TAR_NUMERO, 0, ByVal V_TAR_NUMERO, 0)

'Llamando al servicio

rc = tpcall("SVC_MOVTOS_ABO", buffer.ptr&, 0, buffer.ptr&, lon, 0)
If rc < 0 Then
    Error_aplicacion = gettpurcode()
    Select Case Error_aplicacion
    Case 1
        MsgBox "No existe el número de Tarjeta."
    Case Else
        TuxError ("Error en la Consulta.")
    End Select
    Exit Sub
End If

'Verifica el número de Ocurrencias


Ocurrencias = Foccur32(buffer.ptr, ABO_MONTO)

If Ocurrencias = 0 Then
    MsgBox "No existen cargos."
    tpfree (buffer.ptr)
    Exit Sub
End If

Gd_abonos.Rows = Ocurrencias + 1

    For Ocurrencia = 0 To (Ocurrencias - 1)
        lon = Flen32(buffer.ptr, ABO_MONTO, Ocurrencia)
        rc = Fget32(buffer.ptr, ABO_MONTO, Ocurrencia, V_ABO_MONTO, lon)
        'Imprime el valor
        Gd_abonos.Row = Ocurrencia + 1
        Gd_abonos.Col = 0
        Gd_abonos.Text = V_ABO_MONTO

        lon = Flen32(buffer.ptr, ABO_FEC, Ocurrencia)
        rc = Fget32(buffer.ptr, ABO_FEC, Ocurrencia, ByVal V_ABO_FEC, lon)
        'Imprime el valor
        Gd_abonos.Row = Ocurrencia + 1
        Gd_abonos.Col = 1
        Gd_abonos.Text = V_ABO_FEC

        lon = Flen32(buffer.ptr, ABO_OBSER, Ocurrencia)
        rc = Fget32(buffer.ptr, ABO_OBSER, Ocurrencia, ByVal V_ABO_OBSER, lon)
        'Imprime el valor
        Gd_abonos.Row = Ocurrencia + 1
        Gd_abonos.Col = 2
        Gd_abonos.Text = V_ABO_OBSER
Next

tpfree (buffer.ptr)

End Sub
```

Servidores

Los Servidores Aplicativos son programas que contiene funciones llamadas Servicios, estos Servicios contienen las reglas del negocio del Prototipo y serán invocados por el Cliente o por otros Servicios.

Los Servidores que contienen Servicios que accesan a la Base de Datos realizan como primera actividad la conexión a la misma, en el Prototipo la conexión se realiza mediante el uso de SQL inmerso en C:

```
EXEC SQL CONNECT TO tesis USER sa;
```

Las operaciones como Inserciones o Consultas sobre la Base de datos se realizan también por SQL inmerso, por ejemplo:

```
EXEC SQL SELECT pat_id INTO :patron.pat_id FROM patrones
    WHERE pat_nom = :patron.pat_nom
    AND pat_tel = :patron.pat_tel;
```

Dentro del prototipo existen 8 Servidores Aplicativos llamados:

- SVR_INSERTA
- SVR_AUTORIZA
- SVR_CATALOGO
- SVR_CONSULTA
- SVR_CARGO
- SVR_ABONO
- SVR_MOVTOS_CAR
- SVR_MOVTOS_ABO

Los Servicios que contiene cada Servidor son:

SVR_INSERTA

- SVC_INSERTA que se encarga de ingresar a la Base de Datos los nuevos tarjetahabientes.

SVR_AUTORIZA

- SVC_AUTORIZA el cual válida si un prospecto puede convertirse en tarjetahabiente.

SVR_CATALOGO

- SVC_CATALOGO el cuál recupera toda la información de los catálogos de la aplicación (Estados de la república Mexicana, Tipos de Tarjetas, etc.) para que pueden ser desplegados en el Cliente.

SVR_CONSULTA

- SVC_CONSULTA recibe como parámetros algunos datos del cliente usados para consultar la base de datos y recuperar información detallada sobre el cliente consultado.

SVR_CARGO

- SVC_CARGO realiza cargos sobre la tarjeta del cliente.

SVR_ABONO

- SVC_ABONO realiza abonos sobre la tarjeta del cliente.

SVR_MOVTOS_CAR

- SVC_MOVTOS_CAR recibe como parámetros un número de tarjeta y un rango de fechas y devuelve como resultado los cargos realizados con el número de tarjeta especificado en el rango de fechas

SVR_MOVTOS_ABO

- SVC_MOVTOS_ABO recibe como parámetros un número de tarjeta y un rango de fechas y devuelve como resultado los abonos realizados con el número de tarjeta especificado en el rango de fechas

A continuación se lista el código fuente de cada uno de los Servidores Aplicativos:

SVR_INSERTA

```
#include <tux.h>

#define EXISTE 3
#define NO_APROBADA 10


SVC_INSERTA(TPSVCINFO *rqst)
{
        /*      -----------------
                Buffer de trabajo
        */
        FBFR32  *fmlbuf;


        /*      ----------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf = (FBFR32 *)rqst->data;


        /*      ------------------------------------------
                Inicializacion de id's a 0, ya que no se han
                obtenido sino hasta despues de inserciones.
        */
        cli_id = 0;
        pat_id = 0;
        tar_id = 0;


        Fprint32(fmlbuf);

        /*      -----------------
                Datos del cliente
        */
        Fget32(fmlbuf,CLI_APE_PAT,0,(char *)cli_ape_pat,0);
        Fget32(fmlbuf,CLI_APE_MAT,0,(char *)cli_ape_mat,0);
        Fget32(fmlbuf,CLI_NOM,0,(char *)cli_nom,0);
```

```
Fget32(fmlbuf,CLI_RFC,0,(char *)cli_rfc,0);
Fget32(fmlbuf,CLI_DIR,0,(char *)cli_dir,0);
Fget32(fmlbuf,CLI_COL,0,(char *)cli_col,0);
Fget32(fmlbuf,CLI_POB,0,(char *)cli_pob,0);
Fget32(fmlbuf,CLI_ENT_FED,0,(char *)cli_ent_fed,0);
Fget32(fmlbuf,CLI_PAIS,0,(char *)cli_pais,0);
Fget32(fmlbuf,CLI_CP,0,(char *)cli_cp,0);
Fget32(fmlbuf,CLI_TEL,0,(char *)cli_tel,0);
Fget32(fmlbuf,CLI_CIVIL,0,(char *)cli_civil,0);
Fget32(fmlbuf,CLI_FEC_NAC,0,(char *)cli_fec_nac,0);
Fget32(fmlbuf,CLI_ING_MENS,0,(char *)&cli_ing_mens,0);
Fget32(fmlbuf,CLI_SEXO,0,(char *)cli_sexo,0);
Fget32(fmlbuf,CLI_PUESTO,0,(char *)cli_puesto,0);
Fget32(fmlbuf,CLI_FEC_ING,0,(char *)cli_fec_ing,0);


/*      ------------------------------------------
        Datos de la empresa donde trabaja el cliente
*/
Fget32(fmlbuf,PAT_NOM,0,(char *)pat_nom,0);
Fget32(fmlbuf,PAT_DIR,0,(char *)pat_dir,0);
Fget32(fmlbuf,PAT_COL,0,(char *)pat_col,0);
Fget32(fmlbuf,PAT_POB,0,(char *)pat_pob,0);
Fget32(fmlbuf,PAT_ENT_FED,0,(char *)pat_ent_fed,0);
Fget32(fmlbuf,PAT_PAIS,0,(char *)pat_pais,0);
Fget32(fmlbuf,PAT_CP,0,(char *)pat_cp,0);
Fget32(fmlbuf,PAT_TEL,0,(char *)pat_tel,0);
Fget32(fmlbuf,PAT_FAX,0,(char *)pat_fax,0);
Fget32(fmlbuf,PAT_INI_LAB,0,(char *)pat_ini_lab,0);
Fget32(fmlbuf,PAT_FEC_ING,0,(char *)pat_fec_ing,0);


/*      ------------------------------------------------------
        Valido si el cliente no tiene ya una tarjeta de credito
*/
userlog("Chechando si existe el cliente: %s",cli_rfc);

EXEC SQL SELECT cli_tel INTO :cli_tel
        FROM clientes WHERE cli_rfc = :cli_rfc;

if (SQLCODE == 0)
{
        userlog("Este cliente ya existe en la base de datos...");
        tpreturn(TPFAIL,EXISTE,NULL,0,0);
}


/*      ------------------------
        Manejo de errores de SQL
*/
EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();


/*      ------------------------------------------
        Someto a aprobacion la solicitud del cliente
*/
rc = tpcall("SVC_AUTORIZA",(char *)fmlbuf,0,(char **)&fmlbuf,&len,TPNOTRAN);
if (rc == -1)
{
        userlog("ERROR:No se pudo llamar servicio SVC_AUTORIZA");
        userlog("ERROR:%d, %s",tperrno,tpstrerror(tperrno));
        tpreturn(TPFAIL,NO_APROBADA,NULL,0L,0);
}

/*Fprint32(fmlbuf);*/


EXEC SQL SELECT CONVERT(char(11), getdate(), 101) INTO :hoy;

EXEC SQL WHENEVER SQLERROR CONTINUE;
```

```
userlog("Select en patrones");
EXEC SQL SELECT pat_id INTO :pat_id FROM patrones
        WHERE pat_nom = :pat_nom;

if (SQLCODE == 100)
{

        /*      ------------------------------------------
                Inserto datos de patron y obtengo id de patron
        */
        userlog("Insert en patrones");
        EXEC SQL INSERT INTO patrones VALUES
                (
                        :pat_nom,
                        :pat_dir,
                        :pat_col,
                        :pat_pob,
                        :pat_ent_fed,
                        :pat_pais,
                        :pat_cp,
                        :pat_tel,
                        :pat_fax,
                        :pat_ini_lab,
                        /*:patron.pat_fec_ing*/
                        :hoy
                );

        EXEC SQL SELECT pat_id INTO :pat_id FROM patrones
                WHERE pat_nom = :pat_nom;
}

EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();

Fget32(fmlbuf,TAR_ID,0,(char *)&tar_id,0);

/*      ------------------------------------------
        Inserto datos de cliente con id's de patron
        y de tarjeta, y posteriormente obtengo id
        de cliente.
*/
userlog("Insert en clientes");
EXEC SQL INSERT INTO clientes VALUES
        (
                :pat_id,
                :tar_id,
                :cli_ape_pat,
                :cli_ape_mat,
                :cli_nom,
                :cli_rfc,
                :cli_dir,
                :cli_col,
                :cli_pob,
                :cli_ent_fed,
                :cli_pais,
                :cli_cp,
                :cli_tel,
                :cli_civil,
                :cli_fec_nac,
                :cli_ing_mens,
                :cli_sexo,
                :cli_puesto,
                /*:cli_fec_ing*/
                :hoy
        );

userlog("Select en clientes");
EXEC SQL SELECT cli_id INTO :cli_id FROM clientes
        WHERE cli_rfc = :cli_rfc
        AND cli_fec_nac = :cli_fec_nac;
```

```
        /*      ---------------------------------------
                Regreso id's de operacion de autorizacion
                de credito al cliente, asi como datos de
                tarjeta
        */
        userlog("Fchg32 en cli_id");
        Fchg32(fmlbuf,0,CLI_ID,(char *)&cli_id,0);
        userlog("Fchg32 en pat_id");
        Fchg32(fmlbuf,0,PAT_ID,(char *)&pat_id,0);

        Fprint32(fmlbuf);

        userlog("Terminando insercion");
        tpreturn(TPSUCCESS,0,(char *)fmlbuf,0L,0);
}
```

SVR_AUTORIZA

```
#include <tux.h>

#define NO_APROBADA 10

SVC_AUTORIZA(TPSVCINFO *rqst)
{
        /*      ----------------
                Buffer de trabajo
        */
        FBFR32  *fmlbuf;


        /*      -----
                Otras
        */
        int     periodo;
        char    dep[2];


        /*      ---------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf = (FBFR32 *)rqst->data;


        Fprint32(fmlbuf);

        /*      ------------------------
                Manejo de errores de SQL
        */
        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();


        /*      ----------------------------
                Obtencion de datos de cliente
        */
        Fget32(fmlbuf,CLI_PAIS,0,(char *)cli_pais,0);
        Fget32(fmlbuf,CLI_RFC,0,(char *)cli_rfc,0);
        Fget32(fmlbuf,CLI_CP,0,(char *)cli_cp,0);
        Fget32(fmlbuf,CLI_ING_MENS,0,(char *)&cli_ing_mens,0);
        Fget32(fmlbuf,PAT_INI_LAB,0,(char *)pat_ini_lab,0);


        userlog("Pais: [%s]",cli_pais);
        userlog("RFC : [%s]",cli_rfc);
        userlog("CP  : [%s]",cli_cp);
        userlog("Ingreso: [%f]",cli_ing_mens);
        userlog("Inicio : [%s]",pat_ini_lab);
```

```
/*      ----------------------------------
        Proceso de autorizacion de tarjeta
*/
EXEC SQL SELECT CONVERT(char(11), getdate(), 101) INTO :hoy;

periodo = meses(pat_ini_lab,hoy);
userlog("-----------------------------");
userlog("Meses trabajando: %d",periodo);
userlog("Ingresos: %f",cli_ing_mens);
userlog("Fecha Inicio: %s",hoy);
strcpy(tar_fec_ini,hoy);
strcpy(tar_fec_ing,hoy);
strcpy(ven,(char *)ini_ven(hoy));
userlog("Fecha Vencimiento: %s",ven);
strcpy(tar_fec_ven,ven);

if( (periodo > 60) && (cli_ing_mens > 150000.00) )
{
        userlog("INFO: Tarjeta Platino...");
        strcpy(tar_tipo,"P");
        tar_limi_cred = 75000;
        strcpy(tar_status,"I");
}
else if( (periodo > 36) && (cli_ing_mens > 30000.00) )
{
        userlog("INFO: Tarjeta Oro...");
        strcpy(tar_tipo,"O");
        tar_limi_cred = 15000;
        strcpy(tar_status,"I");
}
else if( (periodo > 24) && (cli_ing_mens > 5000.00) )
{
        userlog("INFO: Tarjeta Basica...");
        strcpy(tar_tipo,"B");
        tar_limi_cred = 3500;
        strcpy(tar_status,"I");
}
else
{
        userlog("INFO:La solicitud no fue aprobada.");
        tpreturn(TPFAIL,NO_APROBADA,NULL,0L,0);
}

userlog("Tipo de tarjeta aprobada: [%s]",tar_tipo);

strcpy(tar_numero,(char *)genera_cuenta(cli_rfc));
strcpy(tar_nip,(char *)genera_nip());
strcpy(tar_suc,"100");
strcpy(tar_fec_corte,"10");

EXEC SQL SELECT COUNT(tar_numero) INTO :max FROM tarjetas
        WHERE SUBSTRING(tar_numero,1,10)=:tar_numero;

if (max == 0)
{
        userlog("Asignando digito 0 a numero de cuenta");
        strcat(tar_numero,"0");
}
else
{
        _itoa(max,dep,10);
        userlog("Asignando digito %d a numero de cuenta",dep);
        strcat(tar_numero,dep);
}


userlog("Numero de tarjeta asignado: [%s]",tar_numero);
userlog("Numero de nip asignado    : [%s]",tar_nip);
```

```
        userlog("----------------------------");

        /*      ------------------------------------
                Inserto datos de tarjeta y obtengo id
        */
        userlog("Insert en tarjetas...");
        EXEC SQL INSERT INTO tarjetas VALUES
                (
                        :tar_numero,
                        :tar_suc,
                        :tar_tipo,
                        :tar_nip,
                        :tar_fec_corte,
                        :tar_fec_ini,
                        :tar_fec_ven,
                        :tar_status,
                        :tar_limi_cred,
                        :tar_fec_ing
                );

        userlog("Select en tarjetas...");
        EXEC SQL SELECT tar_id INTO :tar_id FROM tarjetas
                WHERE tar_numero = :tar_numero;


        /*      ------------------------------------
                Envio de datos de operacion de regreso
        */
        userlog("Fchg32's en campos...");
        Fchg32(fmlbuf,TAR_ID,0,(char *)&tar_id,0);
        Fchg32(fmlbuf,TAR_NUMERO,0,(char *)tar_numero,0);
        Fchg32(fmlbuf,TAR_SUC,0,(char *)tar_suc,0);
        Fchg32(fmlbuf,TAR_TIPO,0,(char *)tar_tipo,0);
        Fchg32(fmlbuf,TAR_NIP,0,(char *)tar_nip,0);
        Fchg32(fmlbuf,TAR_FEC_CORTE,0,(char *)tar_fec_corte,0);
        Fchg32(fmlbuf,TAR_FEC_INI,0,(char *)tar_fec_ini,0);
        Fchg32(fmlbuf,TAR_FEC_VEN,0,(char *)tar_fec_ven,0);
        Fchg32(fmlbuf,TAR_STATUS,0,(char *)tar_status,0);
        Fchg32(fmlbuf,TAR_LIMI_CRED,0,(char *)&tar_limi_cred,0);

        userlog("Terminando autorizacion...");

        tpreturn(TPSUCCESS,0,rqst->data,0L,0);
}
```

SVR_CATALOGO

```
#include <tux.h>


SVC_CATALOGO(TPSVCINFO *rqst)
{
        /*      -----------------
                Buffer de trabajo
        */
        FBFR32  *fmlbuf, *f;
        int     i;


        /*      ---------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf = (FBFR32 *)rqst->data;


        f = (FBFR32 *)tpalloc("FML32",NULL,8192);
        if (f == NULL)
```

```
        {
                userlog("Error alojando FML32");
                tpreturn(TPFAIL,0,NULL,0L,0);
        }


        /*      ------------------------
                Manejo de errores de SQL
        */
        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();


        /*      ------------------------------------
                Obtencion de id de catalogo a buscar
        Fget32(fmlbuf,CAT_ID,0,(char *)&descripcion.cat_id,0);
        userlog("Catalogo a buscar: %d",descripcion.cat_id);
        */

        userlog("Select en catalogos...");
        EXEC SQL DECLARE des CURSOR FOR
                SELECT * FROM descripciones;

        EXEC SQL OPEN des;

        EXEC SQL WHENEVER NOT FOUND GOTO nomas;

        for(i=0;;i++)
        {
                EXEC SQL FETCH des INTO
                        :cat_id, :des_id, :des_des;

                Fchg32(f,CAT_ID,i,(char *)&cat_id,0);
                Fchg32(f,DES_ID,i,(char *)des_id,0);
                Fchg32(f,DES_DES,i,(char *)des_des,0);
                /*userlog("Fetch no: %d",i);*/
        }

nomas:
        EXEC SQL CLOSE des;

        /*      ------------------------------------
                Envio de datos de operacion de regreso
        */
        userlog("Terminando catalogos...");

        tpreturn(TPSUCCESS,0,(char *)f,0L,0);
}
```

SVR_CONSULTA

```
#include <tux.h>

#define ESTA_PAT        1
#define ESTA_MAT        1
#define ESTA_TAR        1
#define NOEXISTE        5


/*      ----------------
        Buffer de trabajo
*/
FBFR32 *fmlbuf;
FBFR32 *f;

void llena_cliente(int);
void llena_tarjeta(int);
void llena_patron(int);
```

```
SVC_CONSULTA(TPSVCINFO *rqst)
{
        int     i;
        int     ban1, ban2, ban3;

        ban1= ban2= ban3= 0;


        /*      ---------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf = (FBFR32 *)rqst->data;


        f = (FBFR32 *)tpalloc("FML32",NULL,32768);
        if ( f == NULL)
        {
                userlog("Error alojando memoria...(FML32)");
                tpreturn(TPFAIL,0,(char *)NULL,0L,0);
        }


        /*      ------------------------
                Manejo de errores de SQL
        */
        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();


        /*
                ------------------------------------------------------
                Detectamos datos para determinar criterio de consulta
        */
        rc = Fpres32(fmlbuf,CLI_APE_PAT,0);
        if (rc)
        {
                userlog("Apellido paterno presente...");
                Fget32(fmlbuf,CLI_APE_PAT,0,(char *)cli_ape_pat,0);
                ban1 = ESTA_PAT;
        }

        rc = Fpres32(fmlbuf,CLI_APE_MAT,0);
        if (rc)
        {
                userlog("Apellido materno presente...");
                Fget32(fmlbuf,CLI_APE_MAT,0,(char *)cli_ape_mat,0);
                ban2 = ESTA_MAT;
        }

        rc = Fpres32(fmlbuf,TAR_NUMERO,0);
        if (rc)
        {
                userlog("Numero de tarjeta presente...");
                Fget32(fmlbuf,TAR_NUMERO,0,(char *)tar_numero,0);
                ban3 = ESTA_TAR;
        }


        Fprint32(fmlbuf);

        /*
                ------------------------------------------------
                Determino la consulta en base a los datos obtenidos
        */
        if(ban1 && ban2)
        {
                userlog("Busqueda por paterno y materno...");
                /*
                        ---------------------------------------------
                        Inicio de consulta de cliente en tabla clientes
```

```
        por ambos apellidos
*/
EXEC SQL DECLARE CurCli1 CURSOR FOR
        SELECT cli_id, pat_id, tar_id,
        cli_ape_pat, cli_ape_mat, cli_nom,
        cli_rfc, cli_dir, cli_col,
        cli_pob, cli_ent_fed, cli_pais,
        cli_cp, cli_tel, cli_civil,
        CONVERT(char(11),cli_fec_nac,101),
        cli_ing_mens, cli_sexo, cli_puesto,
        CONVERT(char(11),cli_fec_ing,101)
        FROM clientes WHERE cli_ape_pat = :cli_ape_pat
        AND cli_ape_mat = :cli_ape_mat;

EXEC SQL OPEN CurCli1;

userlog("Select de cliente...");
for(i=0;;i++)
{
        EXEC SQL FETCH CurCli1 INTO :cli_id, :pat_id, :tar_id,
                        :cli_ape_pat, :cli_ape_mat, :cli_nom,
                        :cli_rfc, :cli_dir, :cli_col,
                        :cli_pob, :cli_ent_fed, :cli_pais,
                        :cli_cp, :cli_tel, :cli_civil,
                        :cli_fec_nac, :cli_ing_mens, :cli_sexo,
                        :cli_puesto, :cli_fec_ing;

        if ((i == 0) && (SQLCODE == 100))
        {
                EXEC SQL CLOSE CurCli1;
                userlog("No Existe: [%s %s]",cli_ape_pat,cli_ape_mat);
                tpreturn(TPFAIL,NOEXISTE,(char *)NULL,0L,0);
        }
        else if ((i != 0) && (SQLCODE == 100))
                break;
        else
                llena_cliente(i);

        /*
                ------------------------------------
                Inicio de consulta de patron de cliente
        */
        userlog("Select de patrones...");
        EXEC SQL SELECT
                pat_id, pat_nom, pat_dir,
                pat_col, pat_pob, pat_ent_fed, pat_pais,
                pat_cp, pat_tel, pat_fax,
                CONVERT(char(11),pat_ini_lab, 101),
                CONVERT(char(11),pat_fec_ing,101)
                INTO :pat_id, :pat_nom, :pat_dir,
                :pat_col, :pat_pob, :pat_ent_fed, :pat_pais,
                :pat_cp, :pat_tel, :pat_fax, :pat_ini_lab, :pat_fec_ing
                FROM patrones
                WHERE pat_id = :pat_id;

        llena_patron(i);

        /*
                ------------------------------------------------
                Inicio de consulta de tarjeta de credito de cliente
        */
        userlog("Select de tarjeta...");
        EXEC SQL SELECT
                tar_id, tar_numero, tar_suc,
                tar_tipo, tar_nip,
                CONVERT(char(11),tar_fec_corte,101),
                CONVERT(char(11),tar_fec_ini,101),
                CONVERT(char(11),tar_fec_ven,101),
                tar_status, tar_limi_cred,
                CONVERT(char(11),tar_fec_ing,101)
```

```
                        INTO :tar_id, :tar_numero, :tar_suc,
                        :tar_tipo, :tar_nip, :tar_fec_corte, :tar_fec_ini,
                        :tar_fec_ven, :tar_status, :tar_limi_cred,
                        :tar_fec_ing
                        FROM tarjetas
                        WHERE tar_id = :tar_id;

                llena_tarjeta(i);
        }
        EXEC SQL CLOSE CurCli1;
        userlog("Busqueda en clientes concluida...");

        Fprint32(f);
        tpreturn(TPSUCCESS,0,(char *)f,0L,0);
}
else if(ban1)
{
        userlog("Busqueda por paterno...");
        /*
                ----------------------------------------
                Inicio de consulta de cliente en tabla clientes
                por apellido paterno
        */
        EXEC SQL DECLARE CurCli2 CURSOR FOR
                SELECT  cli_id, pat_id, tar_id,
                cli_ape_pat, cli_ape_mat, cli_nom,
                cli_rfc, cli_dir, cli_col,
                cli_pob, cli_ent_fed, cli_pais,
                cli_cp, cli_tel, cli_civil,
                CONVERT(char(11),cli_fec_nac,101),
                cli_ing_mens, cli_sexo, cli_puesto,
                CONVERT(char(11),cli_fec_ing,101)
                FROM clientes WHERE cli_ape_pat = :cli_ape_pat;

        EXEC SQL OPEN CurCli2;

        for(i=0;;i++)
        {
                EXEC SQL FETCH CurCli2 INTO :cli_id, :pat_id, :tar_id,
                                :cli_ape_pat, :cli_ape_mat, :cli_nom,
                                :cli_rfc, :cli_dir, :cli_col,
                                :cli_pob, :cli_ent_fed, :cli_pais,
                                :cli_cp, :cli_tel, :cli_civil,
                                :cli_fec_nac, :cli_ing_mens, :cli_sexo,
                                :cli_puesto, :cli_fec_ing;

                if ((i == 0) && (SQLCODE == 100))
                {
                        EXEC SQL CLOSE CurCli2;
                        userlog("No Existe: [%s]",cli_ape_pat);
                        tpreturn(TPFAIL,NOEXISTE,(char *)NULL,0L,0);
                }
                else if ((i != 0) && (SQLCODE == 100))
                        break;
                else
                        llena_cliente(i);

                /*
                        ----------------------------------------
                        Inicio de consulta de patron de cliente
                */
                userlog("Select de patrones...");
                EXEC SQL SELECT
                        pat_id, pat_nom, pat_dir,
                        pat_col, pat_pob, pat_ent_fed, pat_pais,
                        pat_cp, pat_tel, pat_fax,
                        CONVERT(char(11),pat_ini_lab, 101),
                        CONVERT(char(11),pat_fec_ing,101)
                        INTO :pat_id, :pat_nom, :pat_dir,
                        :pat_col, :pat_pob, :pat_ent_fed, :pat_pais,
```

```
                        :pat_cp, :pat_tel, :pat_fax, :pat_ini_lab, :pat_fec_ing
                        FROM patrones
                        WHERE pat_id = :pat_id;

                llena_patron(i);

                /*
                        ------------------------------------------------
                        Inicio de consulta de tarjeta de credito de cliente
                */
                userlog("Select de tarjeta...");
                EXEC SQL SELECT
                        tar_id, tar_numero, tar_suc,
                        tar_tipo, tar_nip,
                        CONVERT(char(11),tar_fec_corte,101),
                        CONVERT(char(11),tar_fec_ini,101),
                        CONVERT(char(11),tar_fec_ven,101),
                        tar_status, tar_limi_cred,
                        CONVERT(char(11),tar_fec_ing,101)
                        INTO :tar_id, :tar_numero, :tar_suc,
                        :tar_tipo, :tar_nip, :tar_fec_corte, :tar_fec_ini,
                        :tar_fec_ven, :tar_status, :tar_limi_cred,
                        :tar_fec_ing
                        FROM tarjetas
                        WHERE tar_id = :tar_id;

                llena_tarjeta(i);
        }
        EXEC SQL CLOSE CurCli2;
        userlog("Busqueda en clientes concluida...");

        Fprint32(f);
        tpreturn(TPSUCCESS,0,(char *)f,0L,0);
}
else if(ban2)
{
        userlog("Busqueda por materno...");
        /*
                ------------------------------------------------
                Inicio de consulta de cliente en tabla clientes
                por apellido materno
        */
        EXEC SQL DECLARE CurCli3 CURSOR FOR
                SELECT cli_id, pat_id, tar_id,
                cli_ape_pat, cli_ape_mat, cli_nom,
                cli_rfc, cli_dir, cli_col,
                cli_pob, cli_ent_fed, cli_pais,
                cli_cp, cli_tel, cli_civil,
                CONVERT(char(11),cli_fec_nac,101),
                cli_ing_mens, cli_sexo, cli_puesto,
                CONVERT(char(11),cli_fec_ing,101)
                FROM clientes WHERE cli_ape_mat = :cli_ape_mat;

        EXEC SQL OPEN CurCli3;

        for(i=0;;i++)
        {
                EXEC SQL FETCH CurCli3 INTO :cli_id, :pat_id, :tar_id,
                                :cli_ape_pat, :cli_ape_mat, :cli_nom,
                                :cli_rfc, :cli_dir, :cli_col,
                                :cli_pob, :cli_ent_fed, :cli_pais,
                                :cli_cp, :cli_tel, :cli_civil,
                                :cli_fec_nac, :cli_ing_mens, :cli_sexo,
                                :cli_puesto, :cli_fec_ing;

                if ((i == 0) && (SQLCODE == 100))
                {
                        EXEC SQL CLOSE CurCli3;
                        userlog("No Existe: [%s]",cli_ape_mat);
                        tpreturn(TPFAIL,NOEXISTE,(char *)NULL,0L,0);
```

```
                }
                if ((i != 0) && (SQLCODE == 100))
                        break;
                else
                        llena_cliente(i);

                /*
                        ---------------------------------------
                        Inicio de consulta de patron de cliente
                */
                userlog("Select de patrones...");
                EXEC SQL SELECT
                        pat_id, pat_nom, pat_dir,
                        pat_col, pat_pob, pat_ent_fed, pat_pais,
                        pat_cp, pat_tel, pat_fax,
                        CONVERT(char(11),pat_ini_lab, 101),
                        CONVERT(char(11),pat_fec_ing,101)
                        INTO :pat_id, :pat_nom, :pat_dir,
                        :pat_col, :pat_pob, :pat_ent_fed, :pat_pais,
                        :pat_cp, :pat_tel, :pat_fax, :pat_ini_lab, :pat_fec_ing
                        FROM patrones
                        WHERE pat_id = :pat_id;

                llena_patron(i);

                /*
                        ------------------------------------------------
                        Inicio de consulta de tarjeta de credito de cliente
                */
                userlog("Select de tarjeta...");
                EXEC SQL SELECT
                        tar_id, tar_numero, tar_suc,
                        tar_tipo, tar_nip,
                        CONVERT(char(11),tar_fec_corte,101),
                        CONVERT(char(11),tar_fec_ini,101),
                        CONVERT(char(11),tar_fec_ven,101),
                        tar_status, tar_limi_cred,
                        CONVERT(char(11),tar_fec_ing,101)
                        INTO :tar_id, :tar_numero, :tar_suc,
                        :tar_tipo, :tar_nip, :tar_fec_corte, :tar_fec_ini,
                        :tar_fec_ven, :tar_status, :tar_limi_cred,
                        :tar_fec_ing
                        FROM tarjetas
                        WHERE tar_id = :tar_id;

                llena_tarjeta(i);
        }
        EXEC SQL CLOSE CurCli3;
        userlog("Busqueda en clientes concluida...");

        Fprint32(f);
        tpreturn(TPSUCCESS,0,(char *)f,0L,0);
}
else if(ban3)
{
        userlog("Busqueda por numero de tarjeta...");
        /*
                ------------------------------------------------
                Inicio de consulta de tarjeta por numero de esta
        */
        EXEC SQL DECLARE CurTar CURSOR FOR
                SELECT tar_id, tar_numero, tar_suc,
                tar_tipo, tar_nip,
                CONVERT(char(11),tar_fec_corte,101),
                CONVERT(char(11),tar_fec_ini,101),
                CONVERT(char(11),tar_fec_ven,101),
                tar_status, tar_limi_cred,
                CONVERT(char(11),tar_fec_ing,101)
                FROM tarjetas WHERE tar_numero = :tar_numero;
```

```
EXEC SQL OPEN CurTar;

for(i=0;;i++)
{
        EXEC SQL FETCH CurTar INTO :tar_id, :tar_numero, :tar_suc,
                        :tar_tipo, :tar_nip, :tar_fec_corte,
                        :tar_fec_ini, :tar_fec_ven, :tar_status,
                        :tar_limi_cred, :tar_fec_ing;

        if ((i == 0) && (SQLCODE == 100))
        {
                EXEC SQL CLOSE CurTar;
                userlog("No Existe: [%s]",tar_numero);
                tpreturn(TPFAIL,NOEXISTE,(char *)NULL,0L,0);
        }
        if ((i != 0) && (SQLCODE == 100))
                break;
        else
                llena_tarjeta(i);
}
EXEC SQL CLOSE CurTar;
userlog("Busqueda en tarjetas concluida...");


/*
        ----------------------------------------
        Busqueda de cliente por numero de tarjeta
*/
EXEC SQL DECLARE CurCli CURSOR FOR
        SELECT cli_id, pat_id, tar_id,
        cli_ape_pat, cli_ape_mat, cli_nom,
        cli_rfc, cli_dir, cli_col,
        cli_pob, cli_ent_fed, cli_pais,
        cli_cp, cli_tel, cli_civil,
        CONVERT(char(11),cli_fec_nac,101),
        cli_ing_mens, cli_sexo, cli_puesto,
        CONVERT(char(11),cli_fec_ing,101)
        FROM clientes WHERE tar_id = :tar_id;

EXEC SQL OPEN CurCli;

for(i=0;;i++)
{
        EXEC SQL FETCH CurCli INTO :cli_id, :pat_id, :tar_id,
                        :cli_ape_pat, :cli_ape_mat, :cli_nom,
                        :cli_rfc, :cli_dir, :cli_col,
                        :cli_pob, :cli_ent_fed, :cli_pais,
                        :cli_cp, :cli_tel, :cli_civil,
                        :cli_fec_nac, :cli_ing_mens, :cli_sexo,
                        :cli_puesto, :cli_fec_ing;

        if (SQLCODE == 100)
                break;
        else
                llena_cliente(i);
}
EXEC SQL CLOSE CurCli;
userlog("Busqueda en clientes concluida...");


/*
        ------------------------
        Busqueda de patron por id
*/
EXEC SQL DECLARE CurPat CURSOR FOR
        SELECT pat_id, pat_nom, pat_dir,
        pat_col, pat_pob, pat_ent_fed, pat_pais,
        pat_cp, pat_tel, pat_fax,
        CONVERT(char(11),pat_ini_lab, 101),
        CONVERT(char(11),pat_fec_ing,101)
```

```
                FROM patrones WHERE pat_id = :pat_id;

        EXEC SQL OPEN CurPat;

        for(i=0;;i++)
        {
                EXEC SQL FETCH CurPat INTO :pat_id, :pat_nom, :pat_dir,
                                :pat_col, :pat_pob, :pat_ent_fed, :pat_pais,
                                :pat_cp,    :pat_tel,    :pat_fax,    :pat_ini_lab,
:pat_fec_ing;


                if (SQLCODE == 100)
                        break;
                else
                        llena_patron(i);
        }
        EXEC SQL CLOSE CurPat;
        userlog("Busqueda en patrones concluida...");

        Fprint32(f);
        tpreturn(TPSUCCESS,0,(char *)f,0L,0);
    }
}


void llena_cliente(int j)
{
    /*
        -------------------------
        Envio de datos de cliente
    */
    Fchg32(f,CLI_ID,j,(char *)&cli_id,0);
    Fchg32(f,PAT_ID,j,(char *)&pat_id,0);
    Fchg32(f,TAR_ID,j,(char *)&tar_id,0);
    Fchg32(f,CLI_APE_PAT,j,(char *)cli_ape_pat,0);
    Fchg32(f,CLI_APE_MAT,j,(char *)cli_ape_mat,0);
    Fchg32(f,CLI_NOM,j,(char *)cli_nom,0);
    Fchg32(f,CLI_RFC,j,(char *)cli_rfc,0);
    Fchg32(f,CLI_DIR,j,(char *)cli_dir,0);
    Fchg32(f,CLI_COL,j,(char *)cli_col,0);
    Fchg32(f,CLI_POB,j,(char *)cli_pob,0);
    Fchg32(f,CLI_ENT_FED,j,(char *)cli_ent_fed,0);
    Fchg32(f,CLI_PAIS,j,(char *)cli_pais,0);
    Fchg32(f,CLI_CP,j,(char *)cli_cp,0);
    Fchg32(f,CLI_TEL,j,(char *)cli_tel,0);
    Fchg32(f,CLI_CIVIL,j,(char *)cli_civil,0);
    Fchg32(f,CLI_FEC_NAC,j,(char *)cli_fec_nac,0);
    Fchg32(f,CLI_ING_MENS,j,(char *)&cli_ing_mens,0);
    Fchg32(f,CLI_SEXO,j,(char *)cli_sexo,0);
    Fchg32(f,CLI_PUESTO,j,(char *)cli_puesto,0);
    Fchg32(f,CLI_FEC_ING,j,(char *)cli_fec_ing,0);
}


void llena_patron(int j)
{
    /*
        ------------------------
        Envio de datos de patron
    */
    Fchg32(f,PAT_ID,j,(char *)&pat_id,0);
    Fchg32(f,PAT_NOM,j,(char *)pat_nom,0);
    Fchg32(f,PAT_DIR,j,(char *)pat_dir,0);
    Fchg32(f,PAT_COL,j,(char *)pat_col,0);
    Fchg32(f,PAT_POB,j,(char *)pat_pob,0);
    Fchg32(f,PAT_ENT_FED,j,(char *)pat_ent_fed,0);
    Fchg32(f,PAT_PAIS,j,(char *)pat_pais,0);
    Fchg32(f,PAT_CP,j,(char *)pat_cp,0);
    Fchg32(f,PAT_TEL,j,(char *)pat_tel,0);
```

```
        Fchg32(f,PAT_FAX,j,(char *)pat_fax,0);
        Fchg32(f,PAT_INI_LAB,j,(char *)pat_ini_lab,0);
        Fchg32(f,PAT_FEC_ING,j,(char *)pat_fec_ing,0);
}


void llena_tarjeta(int j)
{
        /*
                -------------------------
                Envio de datos de tarjeta
        */
        Fchg32(f,TAR_ID,j,(char *)&tar_id,0);
        Fchg32(f,TAR_NUMERO,j,(char *)tar_numero,0);
        Fchg32(f,TAR_SUC,j,(char *)tar_suc,0);
        Fchg32(f,TAR_TIPO,j,(char *)tar_tipo,0);
        Fchg32(f,TAR_NIP,j,(char *)tar_nip,0);
        Fchg32(f,TAR_FEC_CORTE,j,(char *)tar_fec_corte,0);
        Fchg32(f,TAR_FEC_INI,j,(char *)tar_fec_ini,0);
        Fchg32(f,TAR_FEC_VEN,j,(char *)tar_fec_ven,0);
        Fchg32(f,TAR_STATUS,j,(char *)tar_status,0);
        Fchg32(f,TAR_LIMI_CRED,j,(char *)&tar_limi_cred,0);
        Fchg32(f,TAR_FEC_ING,j,(char *)tar_fec_ing,0);
}
```

SVR_CARGO

```
#include <tux.h>

SVC_CARGO(TPSVCINFO *rqst)
{
        /* ----------------------------
                Buffer de trabajo
        */
        FBFR32  *fmlbuf;
        float   lf_saldo_temporal, lf_saldo_favor, lf_saldo_deudor;
        float   lf_cargo_total, lf_car_monto_total;
        float   lf_abo_monto_total;
        float   lf_credito_disponible;
        int     i;

        /* ----------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf=(FBFR32 *)rqst->data;

        lf_cargo_total = 0;
        lf_car_monto_total = 0;
        lf_saldo_temporal = 0;
        lf_abo_monto_total = 0;
        /* ----------------------------
                Manejo de errores de SQL
        */
        userlog("CARGO->**********      Inicio      de      Transaccion      (CARGO)
**********");

        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();

        /* ----------------------------------------
                Inicializacion de id's a 0, ya que no se han
                obtenido sino hasta despues de inserciones.
        */
        strcpy(tar_numero,"");

        /* ----------------------------
                Datos de tarjeta
        */
        Fget32(fmlbuf,TAR_NUMERO,0,(char *)tar_numero,0);
```

```
        /* ------------------------------
                Obtiene datos de cargo
        */
        Fget32(fmlbuf,CAR_MONTO,0,(char *)&car_monto,0);
        Fget32(fmlbuf,CAR_FEC,0,(char *)car_fec,0);
/*      Fget32(fmlbuf,CAR_FEC_ING,0,(char *)car_fec_ing,0);*/
        Fget32(fmlbuf,CAR_ESTABLE,0,(char *)car_estable,0);
        Fget32(fmlbuf,CAR_OBSER,0,(char *)car_obser,0);

        /* --------------------------------
                Obtiene el id de la tarjeta
        */

        EXEC SQL SELECT tar_id,tar_limi_cred INTO
                        :tar_id, :tar_limi_cred FROM tarjetas
                        WHERE tar_numero = :tar_numero;

        if (SQLCODE == 100)
        {
                userlog("CARGO->Registro no encontrado");
                tpreturn(TPFAIL,1,rqst->data,0L,0);
        }


        userlog("CARGO->ID de Tarjeta: %d",tar_id);
        userlog("CARGO->Limite de Credito: %2.f",tar_limi_cred);

        EXEC SQL DECLARE cursor_abonos CURSOR FOR
                SELECT abo_monto FROM abonos
                WHERE tar_id =:tar_id;
        EXEC SQL OPEN cursor_abonos;
        lf_abo_monto_total = 0;
        i=1;
        while (SQLCODE == 0)
        {
                EXEC SQL FETCH cursor_abonos INTO :abo_monto_tempo;
                if (SQLCODE == 0)
                {
                        lf_abo_monto_total += abo_monto_tempo;
                        i++;
                }
        }
        EXEC SQL CLOSE cursor_abonos;

        userlog("CARGO->Abonos: %2.f",lf_abo_monto_total);
        /* ------------------------------------------------------------------------------
*/
        /*
*****************************************************************************/
        /* ------------------------------------------------------------------------------
*/

        EXEC SQL DECLARE cursor_cargos CURSOR FOR
                SELECT car_monto FROM cargos
                WHERE tar_id =:tar_id;

        EXEC SQL OPEN cursor_cargos;
        lf_car_monto_total = 0;
        i=1;
        while (SQLCODE == 0)
        {
                EXEC SQL FETCH cursor_cargos INTO :car_monto_tempo;
                if (SQLCODE == 0)
                {
                        lf_car_monto_total += car_monto_tempo;
                        i++;
                }
        }
        EXEC SQL CLOSE cursor_cargos;
```

```
	userlog("CARGO->Cargos: %2.f",lf_car_monto_total);

	lf_cargo_total = lf_car_monto_total + car_monto;
	if (lf_cargo_total > lf_abo_monto_total)
	{
		lf_saldo_temporal = lf_cargo_total - lf_abo_monto_total;
		if (lf_saldo_temporal > tar_limi_cred)
		{
			userlog("CARGO->Limite de Credito Excedido");
			tpreturn(TPFAIL,2,rqst->data,0L,0);
		}
		else if (lf_saldo_temporal == tar_limi_cred)
		{
			lf_saldo_favor = 0;
			lf_saldo_deudor = lf_saldo_temporal;
			lf_credito_disponible = 0;
		}
		else
		{
			lf_saldo_favor = 0;
			lf_saldo_deudor = lf_saldo_temporal;
			lf_credito_disponible = tar_limi_cred - lf_saldo_temporal;

		}
	}
	else if (lf_cargo_total == lf_abo_monto_total)
	{
		lf_saldo_favor = 0;
		lf_saldo_deudor = 0;
		lf_credito_disponible = tar_limi_cred;
	}
	else
	{
		lf_saldo_favor = lf_abo_monto_total - lf_cargo_total;
		lf_saldo_deudor = 0;
		lf_credito_disponible = tar_limi_cred + lf_saldo_favor;
	}

	EXEC SQL SELECT CONVERT(char(11), getdate(), 101) INTO :hoy;
	EXEC SQL INSERT INTO cargos VALUES
		(
			:tar_id,
			:car_monto,
			:car_fec,
			:car_estable,
			:car_obser,
			:hoy
		);
	userlog("CARGO->Saldo Favor: %2.f",lf_saldo_favor);
	userlog("CARGO->Saldo Deudor: %2.f",lf_saldo_deudor);
	userlog("CARGO->Credito Disponible: %2.f",lf_credito_disponible);
	userlog("CARGO->**********	Fin	de	Transaccion	(CARGO)
**********");
	tpreturn(TPSUCCESS,0,rqst->data,0L,0);
}
```

SVR_ABONO

```
#include <tux.h>

SVC_ABONO(TPSVCINFO	*rqst)
{

	/* -------------------------------
		Buffer de trabajo
	*/
	FBFR32 *fmlbuf;
	float  lf_saldo_favor=0.0;
```

```
        float   lf_saldo_deudor=0.0;
        int     i;
        float   lf_credito_disponible=0.0;
        float   lf_abo_monto_total=0.0;
        float   lf_car_monto_total=0.0;

        /* ------------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf=(FBFR32 *)rqst->data;

        /* ------------------------------
                Manejo de errores de SQL
        */

        abo_monto_tempo = 0;
        abo_monto_tempo = 0;
        abo_id=0;
        tar_limi_cred=0.0;

        userlog("******************    INICIO    DE    TRANSACCION    (A    B    O    N    O)
**********************");
        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();

        /* ------------------------------------------------------
                Inicializacion de id's a 0, ya que no se han
                obtenido sino hasta despues de inserciones
        */
        strcpy(tar_numero,"");

        /* ------------------------------
                Datos de tarjeta
        */
        Fget32(fmlbuf,TAR_NUMERO,0,(char *)tar_numero,0);
        userlog("ABONO:Numero de tarjeta->[%s]",tar_numero);

        /* ---------------------------------------
                Obtiene el id de la tarjeta
        */
        EXEC SQL SELECT tar_id,tar_limi_cred INTO
                        :tar_id, :tar_limi_cred FROM tarjetas
                        WHERE tar_numero = :tar_numero;

        if (SQLCODE == 100)
        {
                userlog("ABONO->Registro no encontrado");
                tpreturn(TPFAIL,1,rqst->data,0L,0);
        }
        userlog("ABONO:tar_id->[%d]",tar_id);
        userlog("ABONO:tar_limi_cred->[%2.f]",tar_limi_cred);

        /* ---------------------------------------------
                Obtiene los datos de cargo
        */
        Fget32(fmlbuf,ABO_MONTO,0,(char *)&abo_monto,0);
        Fget32(fmlbuf,ABO_FEC,0,(char *)abo_fec,0);
        Fget32(fmlbuf,ABO_SUC,0,(char *)abo_suc,0);
        Fget32(fmlbuf,ABO_OBSER,0,(char *)abo_obser,0);

        EXEC SQL SELECT CONVERT(char(11), getdate(), 101) INTO :hoy;

        EXEC SQL INSERT INTO abonos VALUES
        (
                :tar_id,
                :abo_monto,
                :abo_fec,
                :abo_suc,
                :abo_obser,
                :hoy
        );
```

```
/* ------------------------------------------------------
       Obtiene el total de los abonos realizados
       durante ese periodo
*/
EXEC SQL DECLARE cursor_abonos CURSOR FOR
       SELECT abo_monto FROM abonos
       WHERE tar_id =:tar_id;

EXEC SQL OPEN cursor_abonos;
i=1;
while (SQLCODE == 0)
{
       EXEC SQL FETCH cursor_abonos INTO :abo_monto_tempo;
       if (SQLCODE == 0)
       {
              lf_abo_monto_total += abo_monto_tempo;
              i++;
       }
}
EXEC SQL CLOSE cursor_abonos;
userlog("ABONO->Abonos total: %2.f",lf_abo_monto_total);

/* ------------------------------------------------------
       Obtiene el total de los cargos realizados
       durante ese periodo
*/
EXEC SQL DECLARE cursor_cargos CURSOR FOR
       SELECT car_monto FROM cargos
       WHERE  tar_id =:tar_id;

EXEC SQL OPEN cursor_cargos;
i=1;
while (SQLCODE == 0)
{
       EXEC SQL FETCH cursor_cargos INTO :car_monto_tempo;
       if (SQLCODE == 0)
       {
              lf_car_monto_total+= car_monto_tempo;
              i++;
       }
}
EXEC SQL CLOSE cursor_cargos;
userlog("ABONO->Cargos total: %2.f",lf_car_monto_total);

if (lf_abo_monto_total > lf_car_monto_total)
{
       lf_saldo_favor = lf_abo_monto_total - lf_car_monto_total;
       lf_saldo_deudor = 0;
       lf_credito_disponible = tar_limi_cred + lf_saldo_favor;
}
else if (lf_abo_monto_total == lf_car_monto_total)
{
       lf_saldo_favor = 0;
       lf_saldo_deudor = 0;
       lf_credito_disponible = tar_limi_cred;
}
else
{
       lf_saldo_favor = 0;
       lf_saldo_deudor = lf_car_monto_total - lf_abo_monto_total;
       lf_credito_disponible = tar_limi_cred - lf_saldo_deudor;
}
userlog("ABONO->Limite de credito: %2.f",tar_limi_cred);
userlog("ABONO->Saldo favor: %2.f",lf_saldo_favor);
userlog("ABONO->Saldo deudor: %2.f",lf_saldo_deudor);
userlog("ABONO->Credito Disponible: %2.f",lf_credito_disponible);
userlog("****************** FIN DE TRANSACCION (A B O N O) **********************");

tpreturn(TPSUCCESS,0,rqst->data,0L,0);
```

```
}
```

SVR_MOVTOS_CAR

```
#include <tux.h>

SVC_MOVTOS_CAR(TPSVCINFO *rqst)
{
        /* ----------------------------
                Buffer de trabajo
        */
        FBFR32  *fmlbuf;
        int     i,index;

        /* ----------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf=(FBFR32 *)rqst->data;

        /* ----------------------------
                Manejo de errores de SQL
        */
        userlog("MOVTOS->****************** Inicio de Transaccion **********************");

        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();

        /* ------------------------------------------
                Inicializacion de id's a 0, ya que no se han
                obtenido sino hasta despues de inserciones.
        */
        strcpy(tar_numero,"");

        /* ----------------------------
                Datos de tarjeta
        */
        Fget32(fmlbuf,TAR_NUMERO,0,(char *)tar_numero,0);
        Fget32(fmlbuf,CAR_FEC,0,(char *)car_fec1,0);
        Fget32(fmlbuf,CAR_FEC,1,(char *)car_fec2,0);

        userlog("MOVTOS->tar_numero: %s",tar_numero);
        userlog("MOVTOS->car_fec1: %s",car_fec1);
        userlog("MOVTOS->car_fec2: %s",car_fec2);

        /* --------------------------------
                Obtiene el id de la tarjeta
        */

        EXEC SQL SELECT tar_id INTO
                        :tar_id FROM tarjetas
                        WHERE tar_numero = :tar_numero;

        if (SQLCODE == 100)
        {
                userlog("MOVTOS->Registro no encontrado");
                tpreturn(TPFAIL,1,rqst->data,0L,0);
        }

        userlog("MOVTOS->Id de la tarjeta: %d",tar_id);

        /* ----------------------------
                Obtener los movimientos de CARGOS
                de la fecha de corte al
                dia de hoy
        */

        EXEC SQL DECLARE cursor_movimientos CURSOR FOR
                SELECT car_monto, CONVERT(char(11),car_fec,101), car_obser
```

```
            FROM cargos WHERE tar_id = :tar_id AND
            (car_fec BETWEEN :car_fec1 AND :car_fec2);

        EXEC SQL OPEN cursor_movimientos;

        EXEC SQL WHENEVER NOT FOUND GOTO termina;

        for(index=0;;index++)
        {
                EXEC SQL FETCH cursor_movimientos
                        INTO :car_monto, :car_fec, :car_obser;
                userlog("MOVTOS->SQLCODE: %d",SQLCODE);
                userlog("MOVIMIENTOS->Monto: %2.f",car_monto);
                userlog("MOVIMIENTOS->Fecha del monto: %s",car_fec);
                userlog("MOVIMIENTOS->Observaciones: %s",car_obser);
                Fchg32(fmlbuf,CAR_MONTO,index,(char *)&car_monto,0);
                Fchg32(fmlbuf,CAR_FEC,index,(char *)car_fec,0);
                Fchg32(fmlbuf,CAR_OBSER,index,(char *)car_obser,0);

        }


termina:
        EXEC SQL CLOSE cursor_movimientos;
        tpreturn(TPSUCCESS,0,rqst->data,0L,0);
}
```

SVR_MOVTOS_ABO

```
#include <tux.h>

SVC_MOVTOS_ABO(TPSVCINFO *rqst)
{
        /* ----------------------------
                Buffer de trabajo
        */
        FBFR32  *fmlbuf;
        int     i,index;

        /* ----------------------------
                Mapeamos los datos de buffer
        */
        fmlbuf=(FBFR32 *)rqst->data;

        /* ----------------------------
                Manejo de errores de SQL
        */
        userlog("MOVTOS->****************** Inicio de Transaccion **********************");

        EXEC SQL WHENEVER SQLERROR CALL ErrorHandler();

        /* ------------------------------------------
                Inicializacion de id's a 0, ya que no se han
                obtenido sino hasta despues de inserciones.
        */
        strcpy(tar_numero,"");

        /* ----------------------------
                Datos de tarjeta
        */
        Fget32(fmlbuf,TAR_NUMERO,0,(char *)tar_numero,0);
        Fget32(fmlbuf,ABO_FEC,0,(char *)abo_fec1,0);
        Fget32(fmlbuf,ABO_FEC,1,(char *)abo_fec2,0);

        userlog("MOVTOS->tar_numero: %s",tar_numero);
        userlog("MOVTOS->abo_fec1: %s",abo_fec1);
        userlog("MOVTOS->abo_fec2: %s",abo_fec2);
```

```
        /* --------------------------------
                Obtiene el id de la tarjeta
        */

        EXEC SQL SELECT tar_id INTO
                        :tar_id FROM tarjetas
                        WHERE tar_numero = :tar_numero;

        if (SQLCODE == 100)
        {
                userlog("MOVTOS->Registro no encontrado");
                tpreturn(TPFAIL,1,rqst->data,0L,0);
        }

        userlog("MOVTOS->Id de la tarjeta: %d",tar_id);

        /* ----------------------------
                Obtener los movimientos de CARGOS
                de la fecha de corte al
                día de hoy
        */

        EXEC SQL DECLARE cursor_movimientos CURSOR FOR
                SELECT abo_monto, CONVERT(char(11),abo_fec,101), abo_obser
                FROM abonos WHERE tar_id = :tar_id AND
                (abo_fec BETWEEN :abo_fec1 AND :abo_fec2);

        EXEC SQL OPEN cursor_movimientos;

        EXEC SQL WHENEVER NOT FOUND goto termina;

        for(index=0;;index++)
        {
                EXEC SQL FETCH cursor_movimientos
                        INTO :abo_monto, :abo_fec, :abo_obser;
                userlog("MOVTOS_ABONO->Monto: %2.f",abo_monto);
                userlog("MOVTOS_ABONO->Fecha del monto: %s",abo_fec);
                userlog("MOVTOS_ABONO->Observaciones: %s",abo_obser);
                Fchg32(fmlbuf,ABO_MONTO,index,(char *)&abo_monto,0);
                Fchg32(fmlbuf,ABO_FEC,index,(char *)abo_fec,0);
                Fchg32(fmlbuf,ABO_OBSER,index,(char *)abo_obser,0);

        }

termina:
        EXEC SQL CLOSE cursor_movimientos;
        tpreturn(TPSUCCESS,0,rqst->data,0L,0);
}
```

Configuración del Prototipo

La configuración del Prototipo se realiza a través de la modificación de los parámetros de la Herramienta Integradora de Sistemas (BEA TUXEDO).

Los parámetros de BEA TUXEDO se modifican a través de la construcción de archivos de configuración que definen la manera en que el Prototipo va a relacionarse con el sistema operativo, la manera en que va a trabajar y los recurso que puede utilizar. La información de la configuración

reside en una Base de Información Administrativa (MIB, Management Information Base) que es accesible vía programación.

El MIB incluye la siguiente información sobre la configuración del Prototipo:

- La información sobre los atributos globales del Prototipo como el nivel de seguridad, cuando aplica el balanceo de cargas, y la definición de los recursos necesarios para iniciar la aplicación y las acciones a tomar en caso de alguna falla.

- La definición de los equipos de computo que participan en la aplicación y la especificación de que servicios residen en que equipo.

- Los grupo de recurso que los equipos de computo pueden compartir, tales como el administrador de transacciones.

- La cantidad de servicios que están disponibles en cada uno de los equipos.

- Los atributos al nivel de servicio como el factor de carga (la cantidad aproximada de procesamiento por servicio) y la prioridad del servicio con respecto a otros servicios.

El archivo de configuración que utiliza este Prototipo se llama UBB y se lista a continuación:

```
*RESOURCES
IPCKEY          50000

DOMAINID        Tesis
MASTER          App
MAXACCESSERS    70
MAXSERVERS      50
MAXSERVICES     50
MODEL           SHM
LDBAL           N

*MACHINES
WOOKIE          LMID=App
                APPDIR="d:\tesis\app"
```

```
                TUXCONFIG="d:\tesis\app\tuxconfig"
                TUXDIR="d:\tuxedo"
                TLOGDEVICE="d:\tesis\app\TLOG"
                TLOGNAME=TLOG
                TLOGSIZE=100
                MAXWSCLIENTS=30

*GROUPS
GROUP1
        LMID=App GRPNO=1

*SERVERS
DEFAULT:
        CLOPT="-A"

SVR_INSERTA     SRVGRP=GROUP1 SRVID=1 MIN=1 MAX=10 MAXGEN=2
SVR_AUTORIZA    SRVGRP=GROUP1 SRVID=11 MIN=1 MAX=10 MAXGEN=2
SVR_CATALOGO    SRVGRP=GROUP1 SRVID=21 MIN=1 MAX=10 MAXGEN=2
SVR_CONSULTA    SRVGRP=GROUP1 SRVID=31 MIN=1 MAX=10 MAXGEN=2

WSL      SRVGRP=GROUP1 SRVID=50
        CLOPT="-A -- -n //WOOKIE:2000 -m 1 -M 2 -x 30"

SVR_CARGO       SRVGRP=GROUP1 SRVID=51 MIN=1 MAX=10 MAXGEN=2
SVR_ABONO       SRVGRP=GROUP1 SRVID=61 MIN=1 MAX=10 MAXGEN=2
SVR_MOVTOS_CAR SRVGRP=GROUP1 SRVID=71 MIN=1 MAX=10 MAXGEN=2
SVR_MOVTOS_ABO SRVGRP=GROUP1 SRVID=81 MIN=1 MAX=10 MAXGEN=2

*SERVICES
SVC_INSERTA     AUTOTRAN=Y      TRANTIME=20
SVC_AUTORIZA    AUTOTRAN=Y      TRANTIME=20
SVC_CATALOGO    AUTOTRAN=Y      TRANTIME=20
SVC_CONSULTA    AUTOTRAN=Y      TRANTIME=20
SVC_CARGO       AUTOTRAN=Y      TRANTIME=20
SVC_ABONO       AUTOTRAN=Y      TRANTIME=20
SVC_MOVTOS_CAR AUTOTRAN=Y      TRANTIME=20
SVC_MOVTOS_ABO AUTOTRAN=Y      TRANTIME=20
```

En el archivo se pueden apreciar varias secciones y dentro de cada sección se definen los parámetros de la aplicación.

En la sección RESOURCES se declaran los parámetros globales del Prototipo como el identificador único de la aplicación.

En MACHINES se definen los identificadores de cada equipo de computo y se determina la ubicación de los archivos utilizados por el Prototipo.

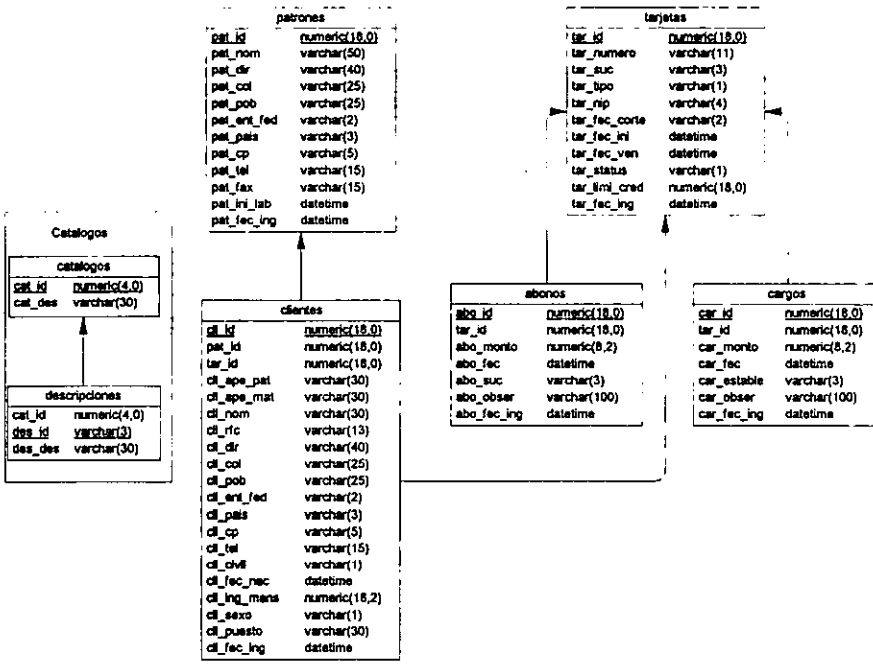
En GROUPS se agrupan los Servidores Aplicativos en relación con los equipos, debe existir un grupo por cada equipo definido

En SERVERS se definen los nombres de los Servidores Aplicativos y la cantidad de cada uno de ellos, en esta sección se incluye un Servidor especial llamado WSL que se encarga de establecer y mantener la comunicación con los Clientes.

En SERVICES se declaran los Servicios del Prototipo, su prioridad y su factor de carga.

Modelo de Lógico

A continuación se muestra el Modelo Lógico usado en el Prototipo.



Conclusiones

En este apartado se presentan los resultados obtenidos durante el desarrollo de la tesis.

Conclusiones

Durante el desarrollo de la presente tesis llegamos a las siguientes conclusiones:

1. El Desarrollo de sistemas bajo la arquitectura c/s de tres capas, siendo la capa intermedia un monitor de transacciones, necesita de un diseño basado en el conocimiento de las ventajas y desventajas de esta misma. Por lo cual, es necesario personal técnico altamente capacitado en las herramientas usadas en este tipo de arquitectura.

   Por experiencia profesional, nos hemos dado cuenta que una deficiencia durante la fase del diseño provoca dificultades que inician en la etapa de desarrollo y que se vuelven más graves en la etapa de implantación del sistema.

   Por ejemplo: Un sistema desarrollado para establecer transacciones en línea que cuenta con infraestructura deficiente de comunicaciones, tiene altas probabilidades de generar la perdida de mensajes e incurrir en la inestabilidad del sistema. El enfoque en el diseño, debe considerar el uso de un esquema de manejo de colas, esto con el fin de garantizar el proceso de la petición en un momento dado.

   Cualquier error de diseño como el mencionado anteriormente, afecta de manera directa los tiempos de desarrollo del sistema, las pruebas realizadas o a realizar y la calidad del producto entregado al cliente.

   Así mismo, el hecho de utilizar personal con bajo conocimiento en el uso de las herramientas involucradas con la arquitectura, provoca un diseño y código deficiente, que afectará de manera negativa el producto final, así como las actividades posteriores (mantenimiento y administración).

2. Dentro de esta arquitectura, si el diseño y el desarrollo se realiza de manera adecuada, el mantenimiento se convierte en un tarea más fácil de realizar que bajo otras arquitecturas, tales como aplicaciones monolíticas o c/s de dos capas. La facilidad en el mantenimiento se deriva de la división de componentes de la aplicación, donde el mantenimiento de un estos se lleva a cabo de manera independiente.

   Por ejemplo: Un mantenimiento a la presentación de los datos no involucra necesariamente un mantenimiento en el repositorio de los mismos.

3. Encontramos que esta es una arquitectura abierta porque permite integración con diversas herramientas en cualquiera de sus capas. Tales como: diversas Bases de Datos (Oracle, Sybase, Informix, etc.), herramientas de desarrollo gráfico (Visual Basic, Delphi, Power Builder, etc.) y sistemas operativos (Unix, Windows NT, DOS, etc.).
   Al ser una arquitectura abierta, también permite la portabilidad de su componente intermedio a diversos sistemas operativos que ya se han mencionaron, por lo que el desarrollador se enfoca principalmente a la aplicación y no a los esquemas de comunicación entre las plataformas, así como tampoco al manejo del formato de los datos de una plataforma a otra.

4. Los sistemas C/S pueden ser escalados horizontal y verticalmente. Escalar de manera horizontal significa añadir o quitar clientes workstation (usuarios finales) con el menor impacto en el desempeño. Escalar de manera vertical significa poder migrar la aplicación a un servidor con más capacidad de proceso o distribuir el proceso en múltiples servidores.

5. La modularidad de las aplicaciones permite desarrollos más eficientes y fáciles de mantener. Permite que las aplicaciones sean abiertas, por ende, fáciles de extender y de anexar nueva funcionalidad. La modularidad de igual manera permite que las aplicaciones provean de servicios a solicitudes de proceso de una manera más eficiente.

6. Con esta arquitectura se fortalece el manejo de las transacciones del negocio, por lo cual la información se mantiene de una manera consistente.

   Mediante el uso de un monitor de transacciones se facilita la tarea al desarrollador, ya que es este software quien se encarga de la coordinación de las transacciones contra los manejadores de recursos de datos.

7. La administración resulta fácil derivada de la división de la aplicación en componentes, pudiéndose monitorear el comportamiento de estos de una manera independiente, a diferencia de otras arquitecturas, tales como, c/s de dos capas y monolíticas.

8. Una aplicación desarrollada bajo una arquitectura c/s de tres capas puede ser clasificada como de alta disponibilidad al combinar el uso de los siguientes elementos: encolamientos en disco, en caso de no haber disponibilidad de la aplicación; soporte de "two phase commit" contra las bases de datos, separación de la lógica del negocio de equipos clientes y servidores, así como el aprovechamiento de sus capacidades de cómputo y la distribución inteligente de las mismas.

Finalmente podemos concluir que el éxito de un sistema desarrollado e implantado bajo esta arquitectura, depende fundamentalmente del conocimiento y buena aplicación de esta tecnología.

Glosario

A

Aplicación a la Interfase de Manejador de Transacciones (ATMI – Application to Transaction Manager Interface).

API de BEA Tuxedo, que comprende 30 llamadas simples que funcionan como macros para la construcción e implantanción de aplicaciones OLTP. ATMI ha sido adoptado por el grupo 'The Open Group'.

Atomicidad (Atomicity).

Una de las propiedades ACID; toda o nada de la transacción debe de ocurrir. Si no todas las partes de la transacción pueden ocurrir de manera exitosa, todos los efectos de la transacción deben de ser desechos o aplicar una operación de 'rollback'.

Auntenticar (Authenticate).

Determinar de manera confiable la identidad de un usuario, generalmente utilizando un password u una serie de passwords. Una vez autenticado, la identidad puede ser comparada contra las tablas de autorización de servicios y objetos. Esta comparación generalmente tiene lugar en la lista de control de acceso.

Autorizar (Authorize).

Otorgar a un usuario (o proceso) permiso o acceso a los objetos y/o servicios.

Aislamiento (Isolation).

Una de las propiedades ACID; una transacción debe de ser distinguida y discreta de otras transacciones. Ninguna transacción en ejecución debe de interferir de manera concurrente la ejecución de otra transacción.

B

BEA Tuxedo.

Una maquinaria de middleware robusto para el desarrollo e implantación de aplicaciones empresariales. Maneja el proceso de transacciones distribuidas, mecanismo de mensajes en las aplicaciones, y un completo complemento de servicios necesarios para construir y ejecutar aplicaciones de negocio críticas.

Broker.

En el paradigma de comunicación del tipo publicar y suscribir, el broker es un sistema cuyo rol es mantener las suscripciones y hacer que las acciones de los suscriptores ocurran cuando los eventos sean notificados.

Buffer.

Un conjunto de páginas en memoria utilizadas para accesar páginas de disco en uso y recientemente usadas.

Bulletin Board.

Estructura de memoria distribuida, parcialmente replicada, para mantener información para el nombramiento de servicios, manejo de transacciones, y ejecución en BEA Tuxedo.

Balanceo de Cargas (Load Balancing).

La habilidad de un sistema de asegurar el máximo desempeño al ubicar automáticamente el servidor más disponible para una petición, y enviando la petición a dicho servidor, o a la cola de mensajes del servidor para su proceso.

Base Administrativa de Información (MIB - Management Information Base).

Una base de datos con información acerca de dispositivos en red. Un MIB puede incluir datos, tales como el estado de un dispositivo, estadísticas de rendimiento, detección de eventos, y alertas de errores y problemas.

C

Cache.

Un conjunto de memoria que contiene copias de las partes accesadas frecuentemente, pertenecientes a una memoria más grande.

Cliente (Client).

Una aplicación (proceso) que requiere servicios de otras aplicaciones.

Cliente/Servidor (Client/Server).

Una relación entre dos procesos, un cliente y un servidor, en la cual, el cliente hace una petición al servidor. Es posible para el servidor en algún momento hacer una petición al cliente e invertir los papeles.

Cliente/Servidor de Tres-Capas (Three-Tier Client/Server).

Una implantación de Cliente/Servidor de N-Capas.

Commit.

La declaración o proceso de actualizar las operaciónes y mensajes de una transacción y hacer estos cambios visibles a otras transacciones. Cuando una transacción hace commit, todos los cambios se hacen públicos y durables. Una vez llevada a cabo esta operación, los efectos de la transacción no puedes ser desechos de manera automática.

CORBA (Common Object Request Broker Architecture).

Definida por el grupo Object Management Group. Define los componentes de un objecto abierto y como interactuan dichos componentes. También especifica un conjunto extensivo de servicios para crear y destruir objetos, accesandolos por nombre, guardandolos en zonas persistentes, publicando su estado, y definiendo las relaciones mas adecuadas entre estos.

CRM (Communication Resource Manager).

Canales de comunicación que transportan mensajes, contexto y datos de una transaccion.

Componente (Component).

Parte de una aplicación. Unidad de composición.

Consistencia (Consistency).

Una de las propiedades ACID; los resultados de una transacción deben de poder ser duplicados y predecibles, aún cuando el proceso se distibuya entre varias plataformas.

Constructor (Constructor).

Metodo especial para crear e inicializar nuevas instancias de una clase. Los constructores inicializan los nuevos objetos y sus variables, crea cualesquiera objetos que se requieran, y realiza las operaciones necesarias para que el objeto sea inicializado de manera satisfactoria.

Conversación (Conversation).

Un dialogo sobre una conexión.

Cliente/Servidor de N-Capas (N-Tier C/S).

Un enfoque de desarrollo de aplicaciones que particiona la aplicación lógicamente a través de tres o más ambientes: la computadora de escritorio, uno o mas servidores aplicativos, y un servidor de base de datos. La principal ventaja de C/S de n-capas es que extiende los beneficios de C/S a nivel empresarial. Otras ventajas incluyen, manejo sencillo, escalabilidad, seguridad y alto rendimiento.

Colas (Queues).

Una estructura de datos simple para la entrega de peticiones a los servidores. Los elementos encolados pueden ser ordenados en alguna forma de prioridad. El cliente inserta los elementos en la cola y los servidores obtienen dichos elementos de la cola, tan pronto sea posible o de manera programada.

Commit en dos Fases (Two-Phase Commit (2PC)).

Un método para coordinar una transacción entre más de una base de datos (u otro manejador de datos). Garantiza la integridad de los datos al asegurar que las actualizaciones sean finalizadas en todas las bases de datos participantes, o regresa los datos al estado original en que se encontraban, antes de iniciar la transacción, en caso de que una o más actualizaciones fallaran.

Cliente/Servidor de 2-Capas (Two-Tier Client/Server).

Un enfoque de desarrollo de aplicaciones que divide la aplicación en dos partes y divide el proceso entre una estación de trabajo y un servidor.

D

Disponibilidad (Availability).

Característica de los sistemas transaccionales que contribuye a la ejecución suave, y continua de la operación en escenarios de falla.

Demonio (Daemon).

Un proceso del sistema que procesa y ejecuta en background.

Dominio (Domain).

Una colección de sistemas involucrados en aplicaciones de administración autónoma. Permite llamadas entre aplicaciones o entre dominios y control de transacciones.

Durabilidad (Durability).

Una de las propiedades ACID; el resultado de una transacción debe de ser permanente. La transacción debe también ser robusta, debe de ser capaz de srobevivir a los errores aplicativos y rollback para respuesta a la aplicación.

E

Estado Consistente (Consistent State).

La condición en donde los datos compartidos son correctos y válidos.

Envío de Datos (Data Shipping).

Proceso de una aplicación donde los datos son enviados de regreso a quien origina la petición para su computo.

Encripción (Encrypt).

Generar codigos de acceso para asegurar los datos de extraños o prevenir acceso no autorizado.

G

Gateway.

Cualquier programa que traduce información entre distintos ambientes.

H

Hilo de ejecución (Thread).

Una unidad de ejecución o contexto de ejecución.

I

Interface de Programación de la Aplicación (API – Application Programming Interface).

Un conjunto de código que habilita al desarrollador a iniciar y completar peticiones cliente/servidor dentro de una aplicación.

Interface Común de Gateway (Gateway Common Interface - CGI).

Especificación estándar de protocolo, sobre la cual los servidores de HTTP se comunican con los procesos servidores de la aplicación. Los programas CGI manejan directamente las peticiones del navegador, o invocan a otros programas cuando es necesario.

Instancia (Instance).

Un objeto. Cuando una clase produce un objeto, el objeto es una instancia de la clase.

IIOP (Internet Inter-ORB Protocol).

El protocolo de comunicación definido por OMG (Object Management Group) para la interoperabilidad entre los objetos CORBA.

L

Lista de Acceso de Control (ACL – Access Control List).

Es un mecanismo de autorización asociado con un objeto. Cuando el objeto es accesado, la lista es checada par ver si el cliente esta autorizado o no a realizar la operación que requiere.

Librería de Clases (Class Library).

Un conjunto de herramientas de programación para el cliente.

Llamado de Funciones (Function Shipping).

Llamar a una función (operación) al servidor de datos, en lugar de enviar un conjunto de datos a la función (cliente) – solamente el resultado de la operación en enviado al cliente.

Lenguaje de Definición de Interfase (IDL – Interface Definition Language).

El lenguaje desarrollado por OMG para definir las interfaces de los componentes hacia los clientes. IDL provee de interfaces independientes hacia el sistema operativo – y lenguaje de programación – a todos los componentes y servicios que residen en un bus CORBA.

M

Manejador de Transacciones (TM – Transaction Manager).

El componente del sistema de proceso de transacciones que coordina la ejecución de la transacciones. El manejador de transacciones esta integrado tanto con la aplicación, como con el manejador de datos; de tal manera que puede intercambiar mensajes y manejar los estados de la transacción. En un sistema de

proceso de transacciones distribuido, el manejador de transacciones se puede comunicar con otros manejadores de transacciones para coordinar las transacciones que actualizan varios recursos heterogéneos independientes, o que interopera con otros sistemas de proceso de transacciones heterogéneos. El manejador de transacciones también coordina la recuperación si una falla sucede.

Middleware de Eventos de Negocio (Business Event Middleware).

Middleware orientado a eventos que integra las aplicaciones, al proveer conexión de programa a programa, comunicación, y transferencia de datos a través de la publicación y suscripción.

Mensaje (Message).

Una pieza de código enviada entre componentes, generalmente incluye algunas instrucciones para el componente receptor.

Middleware Orientado a Mensajes (MOM – Message Oriented Middleware).

Un enfoque del middleware para implantar C/S de *n* –capas que combina el encolamiento de mensajes y el paso de mensajes en línea. Los beneficios incluyen soporte para comunicación asíncrona, provee de paralelismo heredado, y escalabilidad entre nodos de red.

Middleware.

Una categoría de software relativamente nueva que describe programas que ligan los componentes de una aplicación. Los acercamientos de un middleware para implantar C/S de n-capas son RPC, MOM, y proceso distribuido de transacciones.

Manejador de Recursos (RM – Resource Manager).

Un subsistema que maneja algunos objetos transaccionales. El manejador de recursos generalmente ofrece servicios a las aplicaciones o algunos otros manejadores de recursos. Un sistema de base de datos transaccional, un manejador de colas, son ejemplos de manejadores de recursos.

Monitor de Proceso de Transacciones (TP Monitor – Transaction Processing Monitor).

Nombre para una clase de productos que provee un ambiente de ejecución de transacciones encima del sistema operativo.

N

Notificación (Broadcast).

Enviar el mismo mensaje a cada nodo en la red.

Nodo (Node).

El punto en el cual la estación de trabajo se conecta al servidor de red.

O

Objeto (Object).

Una instancia concreta de alguna clase.

OMG (Object Management Group).

Un consorcio de más de 650 compañías que definieron CORBA.

ORB (Object Request Broker).

Un bus de objetos que permite a los componentes interoperar entre espacios de dirección, lenguajes, sistemas operativos, y redes; provee de un mecanismo que permite que los componentes intercambien meta datos y se localicen unos a otros.

OLTP (On Line Transaction Processing).

El área de negocios en computo que involucra transacciones de negocio críticas que son procesadas en tiempo real. Estos sistemas requieren una alta capacidad de proceso, y tiempos de respuesta muy cortos.

Ejemplo de aplicaciones OLTP son los puntos de venta (POS – Point of Sale) en una tienda y los cajeros automáticos (ATM – Automated Teller Machine).

OSI TP (Open Systems Interconnect Transaction Processing Protocol).

Un estándar de la industria para habilitar interoperabilidad transaccional entre sistemas compatibles con ISO.

P

Paquetes (Packages).

Una forma de agrupar juntas, clases e interfaces relacionadas.

Particionar (Partition).

Descomponer una aplicación en módulos que pueden a su vez, ser divididos en más de un nodo.

Pipelined Parallelism.

Un paradigma de programación en el cual, el proceso ocurre de manera serial por más de un módulo, de manera que cada nivel del proceso ejecuta alguna parte perteneciente al total de la operación, y pasa los resultados al siguiente nivel. El nivel final responde a quien originó la petición.

Publicación y Suscripción (Publish and Subscribe).

Una forma de proceso distribuido de datos, manejado por los eventos del negocio. El Middleware es usado para activar una serie de procesos entre varias aplicaciones en respuesta a eventos previamente especificados. Los usuarios son automáticamente notificados cuando un evento que los afecta ocurre.

Publicista (Publisher).

En el paradigma de Publicación y Suscripción, el publicista es la aplicación que detecta por la ocurrencia de los eventos del negocio y notifica de dicha ocurrencia.

Proceso de Transacciones por Encolamiento (Queue Transaction Processing).

Proceso de transacciones por medio de su colocación en colas que son atendidas por prioridad por los servidores. El encolamiento es conveniente cuando se quiere llevar a cabo proceso por lotes.

Propiedades ACID (ACID – Properties).

Atomicidad, Consistencia, Aislamiento, y Durabilidad; las características de un sistema OLTP.

Particionamiento de la Aplicación (Aplication Partitioning).

Descomponer una aplicación en elementos que son guardados en diferentes locaciones en la red.

Petición Asíncrona (Asynchronous Request).

Una petición que permite al usuario realizar otro trabajo, mientras la petición es procesada, aumentado la capacidad de proceso paralelo dentro de la aplicación.

Proceso en Lote (Batch).

Procesar de manera asincrona muchos trabajos en un solo grupo.

Proceso Distribuido de Transacciones (DTP – Distributed Transaction Processing).

Un enfoque de middleware que extiende la capacidad de los sistemas OLTP tradicionales entre plataformas y redes. Diseñado para proveer la confiabilidad y alta disponibilidad necesaria para aplicaciónes de alto volumen.

Petición / Respuesta (Request / Response).

Comunicación caracterizada por una sola petición que espera por una sola respuesta. Existen las variaciones síncrona y asíncrona.

Petición Sincrona.

Un petición / respuesta del cliente que espera por una respuesta.

Protocolos TP.

Un conjunto de protocolos estándares por los cuales, los manejadores de proceso de transacciones en sistemas heterogéneos, interoperan.

R

Ruteo dependiente de Datos (Data Dependent Routing).

Un mecanismo en el cual la petición a un servicio, es dirigida a un grupo de proceso en particular, tomando como base uno de los valores contenidos en los datos del mensaje.

Recuperación (Failover).

Tecnicas de manejo de fallas donde los procesos de un componente fallido (hardware o software) son asumidos por otro componente – generalmente una instancia replicada del componente fallido. Un sistema de proceso de transacciones puede implantar tecnicas de recuperación al mantener replicas de las instancias de los componentes, detectar instancias fallidas, y rutear las peticiones y mensajes a aquellos componentes que están 'vivos' y respondiendo de manera normal.

Red Heterogenea (Heterogeneous Network).

Una red que involucra computadoras con una interfase de administración distinta, interfaces de programación, representación de datos, o protocolos de comunicación.

Red Homogenea (Homogeneous Network).

Una red de computadoras que contiene una variedad de computadoras y software.

Recuperación (Recovery).

En los sistemas transaccionales, después de una falla, la habilidad para recuperar el sistema al ultimo estado consistente. En un sistema distribuido, la recuperación podría involucrar el sincronizar de nueva cuenta varios componentes distribuidos. Una vez hecha la recuperación, el proceso puede continuar, y las transacciones abortadas como resultado de la falla pueden ser enviadas para su proceso de nueva cuenta.

RPC (Remote Procedure Call).

Una llamada a un procedimiento local que es ejecutada en un programa no local o espacio de dirección. Permite que la lógica de la aplicación sea dividida entre un cliente y un servidor de tal manera que se aprovechen mejor los recursos disponibles.

Rollback.

Termina una transacción de manera que los recursos actualizados dentro de una transacción regresen a su estado original antes de que la transacción iniciara.

S

Servidor (Server).

Un módulo de software que acepta peticiones de clientes y otros servidores y regresa respuestas.

Servicio (Service).

Una rutina de una aplicación disponible para ser requerida por un cliente en el sistema.

Stored Procedure.

Un conjunto de instrucciones que esta definido, nombrado, o guardado, y ejecutado dentro de un systema de base de datos.

Suscriptor (Subscriber).

En el paradigma de Publicación y Suscripción, el suscriptor es una aplicación que se suscribe a un evento o conjunto de estos, y declara que acción debe de suceder cuando el evento sea notificado.

Sustracción (Substraction).

Una transacción dentro del contexto de otra de mayor prioridad. De las propiedades ACID, tiene atomicidad, consistencia y aislamiento; pero no durabilidad. Aún cuando la sustracción haya sido exitosa, puede ser abortada debido a que la transacción que la precede puede ser abortada.

T

Transacción.

Una construcción lógica, a través de la cual, las aplicaciones realizan operaciones en recursos compartidos (ej. sistema de base de datos). El trabajo hecho en nombre de la transacción esta regido por las propiedades ACID: Atomicidad, Consistencia, Aislamiento, Durabilidad.

Transacción Distribuida (Distributed Transaction).

Una transacción que involucra multiples manejadores transaccionales. En un ambiente de transacciones distribuidas, una aplicación cliente puede enviar peticiones a algunos servidores, pudiendo resultar esto en multiples actualizaciones, en multiples manejadores de recursos. Para completar la transacción, el manejador de la transacción, para cada participante (clientes, servidores, y manejadores de recursos), debe de coordinar el proceso de commit para cada participante dentro de su dominio.

Transacción Conversacional (Conversational Transaction).

Una transacción que tiene intercambio de mensajes, con el cliente como parte del proceso de la transacción. Contrasta con las transacciones normales que sólo tienen un mensaje de petición y uno de respuesta.

Transaccion Global (Global Transaction).

Una transacción que abarca uno o más manejadores de recursos por medio de transacciones locales.

Transparencia de Locación (Location Transparency).

La habilidad para definir un recurso de tal manera que su nombre no implique una dirección específica o locación física.

TCP/IP

Un protocolo de comunicaciones estándar de facto.

The Open Group.

Un consorcio de los vendedores de UNIX, ISVs y usuarios finales de sistemas abiertos formado como resultado de la fusión entre X/Open y Open Software. Fundación que define la portabilidad y estándares de implantación para el ambiente de sistemas abiertos.

TPS (Transaction Per Second).

Un factor de proceso usado en conjunto con el estándar de transacciones definido por TPC. Generalmente el rango implica el máximo número de transacciones que pueden ser procesadas por un sistema, en donde el 90% del tiempo de respuesta de las transacciones es menor a dos segundos.

TPC (Transaction Processing Performance Council).

Un consorcio de (principalmente sistemas y bases de datos) vendedores dedicados a desarrollar y vigilar las especificaciones de pruebas de rendimiento para el computo orientado hacia las transacciones.

TxRPC.

Un API de aplicación de Open Group que involucra la comunicación aplicación-aplicación via RPCs. El soporte de TxRPC permite a los desarrolladores trabajar con código existente en ambientes basados en DCE.

V

Variables de Instancia (Instance Variables).

La definición de los atributos de un objeto.

X

XATMI.

El API de Tuxedo, que fue seleccionado por Open Group como el estándar para la programación de aplicaciones OLTP, y que se nombró XATMI.

BIBLIOGRAFIA

THE TUXEDO SYSTEM.
Juan M. Andrade, Mark T. Carges.
Addison-Wesley Publishing Company, 1996.

THE ESSENTIAL CLIENT/SERVER SURVIVAL GUIDE.
Robert Orfali, Dan Harkey.
Wiley Computer Publishing, 1996.

CLIENT/SERVER COMPUTING.
Patrick Smith, Steve Guengerich.
SAMS, 1994.

BEA TUXEDO ADMINISTRATOR'S GUIDE.
BEA Systems, Inc; 1997.

BEA TUXEDO USER'S GUIDE.
BEA Systems, Inc; 1997.

BEA TUXEDO APPLICATION DEVELOPER'S GUIDE.
BEA Systems, Inc; 1997.

BEA TUXEDO REFERENCE GUIDE.
BEA Systems, Inc; 1997.

BEA TUXEDO PROGRAMMER'S GUIDE.
BEA Systems, Inc; 1997.

COMPUTER NETWORKS: Protocols, Standards, and Interfaces.
Uyless Black.
Prentice-Hall, Inc; 1987.

ORACLE PRO*C USERS GUIDE.
Oracle Inc;1994.

VISUAL BASIC 4 UNLEASHED.
Michel Amundsen, Keith Brophy, Richard Buhrer.
SAMS, 1995.

VISUAL C++ 4 UNLEASHED.
Viktor Toth.
SAMS Publishing, 1996.

COMO PROGRAMAR EN C/C++.
H.M. DEITEL, P.J. DEITEL.
Prentice-Hall, 1995.

SE USING MICROSOFT SQL SERVER 6.5
Steve Wynkoop.
QUE, 1994.

MICROSOFT SQL SERVER 6.5 DBA SURVIVAL GUIDE.
Mark Spenik, Orryn Sledge.
SAMS, 1996.