DESCRIPCIÓN Y UTILIZACIÓN PRÁCTICA DEL PATRÓN DE DISEÑO CALLBACK DISTRIBUIDO EN APLICACIONES DISTRIBUIDAS CON CORBA.

Joaquín Peña, Francisco Leal y Rafael Corchuelo
Informe Técnico
Dpto. Lenguajes y Sistemas Informáticos de la Universidad de Sevilla
Avda. de la Reina Mercedes, s/n. Sevilla, 41.012
E-mail: jpena@lsi.us.es

INDICE

INTRODUCCIÓN. 3

1. FUNDAMENTOS DE CORBA. 4

2. MODELO EN TRES CAPAS. 6

2.1. CAPA CLIENTE. 7
2.2. CAPA LÓGICA O AGENTE. 7
2.3. CAPA SERVIDOR 8
2.4. INTERCONEXIÓN ENTRE CAPAS. 8
2.4.1. Enlace Agente-Servidor. *8*
2.4.2. Enlace Cliente-Agente. *8*

3. PATRON DE DISEÑO CALLBACK DISTRIBUIDO PARA APLICACIONES CORBA. 10

3.1. INTRODUCCIÓN. 10
3.2. DESCRIPCIÓN DEL PATRÓN. 10
3.2.1. Intención. *10*
3.2.2. Motivación. *10*
3.2.3. Aplicabilidad. *11*
3.2.4. Consecuencias: ventajas e inconvenientes. *11*
3.2.5. Implementación. *13*

4. EJEMPLO DE APLICACIÓN. 15

4.1. INTRODUCCIÓN. 15
4.2. DESCRIPCIÓN DE LA APLICACIÓN A DESARROLLAR. 15
4.3. MODELADO DE LA APLICACIÓN. 18
4.4. UTILIZACIÓN DEL PATRÓN DE DISEÑO CALLBACK DISTRIBUÍDO EN LA APLICACIÓN. 21

5. FUNCIONAMIENTO E INSTALACIÓN. 26

5.1. FUNCIONAMIENTO. 26

6. PROBLEMAS Y CONCLUSIONES. 30

6.1. PROBLEMAS. 30
6.2. CONCLUSIONES. 30

7. BIBLIOGRAFÍA 32

INTRODUCCIÓN.

El propósito de este documento es mostrar tanto la estructura como la utilización práctica del patrón de diseño **Callback Distribuido** para aplicaciones CORBA. Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular. En este caso, el contexto que se propone es una aplicación en tres capas con funcionamiento síncrono que será modificada para que se comporte de manera asíncrona utilizando dicho patrón.

Para poder comprender de manera global la aplicación propuesta es necesario conocer algunos conceptos como son CORBA y el modelo en tres capas. CORBA es un middleware que permite conectar clientes y servidores heterogéneos tanto en el lenguaje de programación como en la plataforma de ejecución de una manera transparente.

Por otro lado, la arquitectura en tres capas, que se corresponde al patrón arquitectónico nivel, realiza una división de la aplicación en capas para aislar en cada una de ellas aquellos elementos del sistema que están relacionados entre sí a un alto nivel de abstracción. Las características de una capa concreta no dependen en ningún aspecto del resto de las capas, comportándose cada una de ellas como un elemento independiente del resto.

Este documento se dividirá en 6 capítulos. En el **capítulo primero** se describirá, en términos generales, el estándar CORBA. En el **capítulo segundo** se introducirá el concepto de arquitectura en tres capas. En el **capítulo tercero** se describirá el patrón de diseño **callback distribuido** detallando sus características, ventajas, inconvenientes e implementación. Seguidamente, en el **capítulo cuarto,** se describirá un ejemplo práctico de aplicación al que se le aplicará el patrón **callback**. En el **capítulo quinto** se mostrará el funcionamiento de la aplicación y se destacarán la mejoras conseguidas con la aplicación del patrón. Y por ultimo, en el **capítulo sexto** se hará un recorrido por los problemas más importantes encontrados y las conclusiones inferidas de la realización de este documento.

1. FUNDAMENTOS DE CORBA.

CORBA es una norma, no un producto, que se encarga de especificar la interoperabilidad entre objetos en un entorno distribuido heterogéneo de manera transparente.

CORBA fue impulsada por la OMG (Object Management Group) en 1990. Se propuso una arquitectura basada en componentes con capacidad de interoperabilidad (llevan integrada la conectividad de red) con independencia de la localización. Finalmente, en 1992 la OMG definió el primer ORB (Object Request Broker).

En definitiva, CORBA es un middleware que permite conectar clientes y servidores heterogéneos tanto en el lenguaje de programación como en la plataforma de ejecución. Esto se consigue de forma transparente para los clientes y servidores. Los clientes simplemente invocan un método de un objeto servidor y no tienen que preocuparse de la localización de dicho objeto. Además, la petición se realiza en el lenguaje de programación en que está implementado el cliente aunque el servidor esté realizado en otro distinto. El encargado de la traducción y transporte de la llamada es el ORB (Object Request Broker), una pieza fundamental de la arquitectura de CORBA.

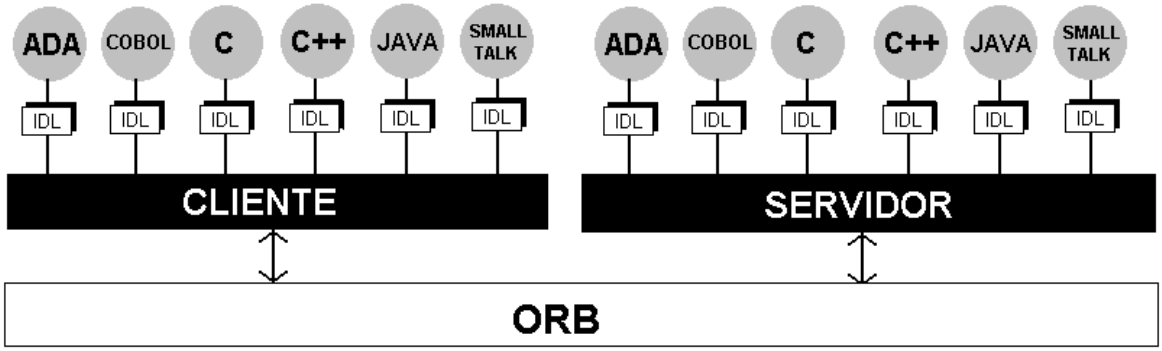


Figura 1. El ORB de CORBA

El ORB es un "bus de objetos" que permite a los objetos hacer llamadas a otros de forma transparente, independientemente de su localización, del sistema operativo donde se ejecuten y del lenguaje en el que están implementados.

Esto se consigue gracias al lenguaje IDL (Interface Definition Language). Mediante IDL se especifican interfaces, consistentes en conjuntos de operaciones que los objetos que actúan como servidores proporcionan a los clientes. Usando un compilador de IDL, se genera el código necesario en un

lenguaje de programación determinado para el lado cliente (los stubs) y el lado servidor (skeletons).

En el lado cliente, el código generado se usa en la implementación del cliente como cualquier librería convencional, que hace ver al cliente el objeto(s) servidor y sus servicios.

En la parte servidora se genera código que recibe las peticiones locales o vía red de los clientes y se encarga de llamar a las funciones del servidor que proporciona la interfaz IDL, cuyas cabeceras/esqueletos también se han generado, debiendo ser implementado por el programador el código de los métodos existentes en la interfaz. Este esquema es similar al de las clásicas Remote Procedure Calls (RPCs).

2. MODELO EN TRES CAPAS.

El modelo en tres capas surge como una mejora sobre la arquitectura clásica en dos capas. Esta basado en la inclusión de una nueva capa o nivel de abstracción en la que se aísla la lógica de administración del sistema del resto de bloques.

El objetivo de esta división es conseguir una mayor independencia entre los elementos que forman parte del sistema, haciendo que cada uno de ellos se comporte de manera transparente y que pueda cambiar sin afectar a los demás. El objetivo es que cada capa se vea como un elemento independiente del resto de capas.

Una de las consecuencias de esta mejora es que la parte cliente puede quedar reducida a una simple interfaz gráfica y la parte servidora a la base de datos y el DBMS. La lógica de negocio entre la parte cliente y la servidor se centralizará en la nueva capa insertada.

Gracias a la versatilidad que proporciona este modelo se podrán utilizar las capas como piezas de un mecano, tomando el número de ellas que sean necesarias y adaptando así el resultado final a distintas necesidades. Además, se podrá reutilizar cada parte tantas veces como sea necesario.

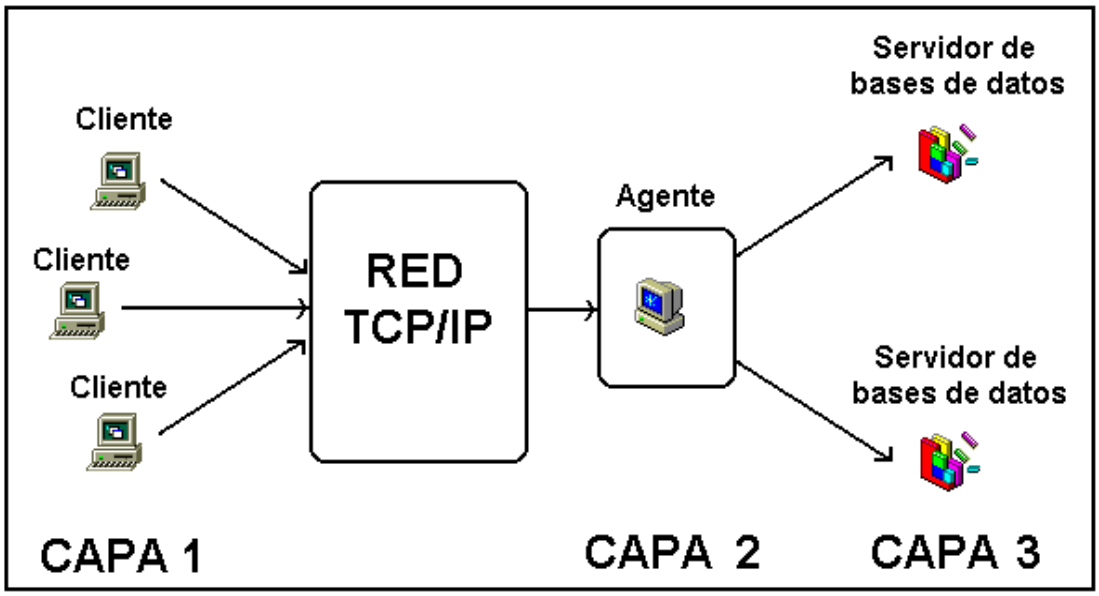


Figura 2. Modelo en tres capas.

A continuación se describirá cada una de las capas que constituyen el modelo.

2.1.Capa Cliente.

La primera capa que se observa en la figura anterior es el cliente. En ella estará implementado el cliente en forma de interfaz gráfica con la que interactuará el usuario. La interfaz permite introducir datos a procesar por la aplicación y muestra los datos que genera como resultado de dicho proceso. La interfaz de usuario puede consistir desde una aplicación independiente hasta un terminal o incluso una pagina web.

La misión de esta capa es únicamente contener la interfaz gráfica para interactuar con el usuario.

Si se tienen distintos tipos de clientes (encargados de sección, cajeros...) con necesidades distintas y plataformas distintas, a la hora de implementarlos la cantidad de código a desarrollar será pequeña, ya que sólo habrá que escribir la interfaz de usuario. Si se utiliza un editor de interfaces gráficas la tarea se hace casi trivial.

2.2. Capa Lógica o Agente.

La nueva capa que introduce el modelo es la capa intermedia, que será la encargada de manejar la lógica de la aplicación. Se denomina capa Agente o capa de Lógica de Negociado.

Esta capa se verá como una especie de interfaz que proporcionará una serie de primitivas y datos a los que será posible acceder. De esta forma, se dispondrá de un conjunto de servicios que podrán ser usados por la capa cliente. El cliente deberá poder ser programado sin necesidad de conocer la implementación interna de la lógica de negociado. El objetivo es que las modificaciones realizadas a la capa lógica mantengan la interfaz (la apariencia externa) intacta, y no desemboquen en la modificación de ninguna de las otras dos capas.

La principal función del agente es la de traducir y adaptar las peticiones del cliente (llamadas a las primitivas que proporciona la capa) a la semántica interna del servidor. En el Agente está centralizado además el acceso de los clientes al servidor, lo que aumenta la seguridad y la eficiencia. De este modo es posible realizar una mejor gestión de la carga del servidor. Se pueden introducir, por ejemplo, varios servidores y balancear la carga entre ellos, con lo que se consigue que él número de clientes pueda aumentar considerablemente.

2.3. Capa Servidor

Esta capa consiste en un gestor de bases de datos (DBMS). Hay que notar que, al existir muchos gestores de bases de datos distintos, cada uno de ellos posee rutinas de acceso diferentes. Esto haría que la capa lógica no fuera totalmente abstracta y dependiera fuertemente del gestor que se utilizara. Como se explicará posteriormente, este problema se puede solucionar gracias a SQL. Al estar estandarizado y existir puentes traductores de sentencias SQL a las propias del gestor, será posible trabajar de manera independiente al gestor de bases de datos utilizado.

2.4. Interconexión entre capas.

Otra de las características del modelo muy a tener en cuenta es la manera en la que se comunican las capas entre sí. La interfaz de usuario se comunicará con el agente, y éste a su vez lo hará con el gestor de bases de datos. Así pues, se tienen dos vías de comunicación que tendrán objetivos y necesidades distintas, por lo que se utilizarán elementos diferentes para cada una de ellas.

2.4.1. Enlace Agente-Servidor.

Los fabricantes de bases de datos suelen proporcionar rutinas de acceso al gestor de la base de datos, pero aunque proporcionan un acceso más rápido no es aconsejable utilizarlas si se quiere mantener la flexibilidad del sistema. Hay otras formas de comunicar al agente con la base de datos de manera que no se tengan que utilizar las rutinas propias de cada gestor. Esto se puede lograr utilizando ODBC o JDBC, gracias a los cuales todas las consultas contra la base de datos se realizarán en SQL. Será ODBC o JDBC, según el caso, el encargado de traducir las peticiones a la semántica propia del gestor que se esté utilizando. Esto permite cambiar el sistema gestor de bases de datos con relativa facilidad, ya que no habrá que modificar prácticamente nada en el código (solo la línea en que se especifica el gestor utilizado en el caso de JDBC) del Agente.

2.4.2. Enlace Cliente-Agente.

Para comunicar el cliente con el agente existen varias posibilidades, entre las que se encuentran java-RMI, DCE, DCOM y CORBA. Si se quiere conseguir un producto flexible, portable y escalable, son estas dos ultimas la más adecuadas. En la actualidad DCOM es un producto que sólo esta disponible para plataformas Windows, y en cambio CORBA lo está para un gran numero de plataformas y lenguajes de programación. CORBA permite utilizar gran variedad de lenguajes de programación (existen implementaciones

de CORBA para C++, JAVA, Smalltalk, Ada 95, COBOL) con lo que el código del cliente y el del agente pueden estar escritos en lenguajes diferentes. Incluso se podrá cambiar el cliente o el agente sin necesidad de reescribir la otra capa.

3. PATRON DE DISEÑO CALLBACK DISTRIBUIDO PARA APLICACIONES CORBA.

3.1. Introducción.

El patrón Callback Distribuido [2] [3] se clasifica dentro de los Patrones de Diseño, en el grupo de los Patrones de Comportamiento. El objetivo de este patrón es reducir el tiempo que emplea un cliente esperando el resultado de una operación remota, convirtiendo una operación síncrona en asíncrona tal y como se observa en la figura.

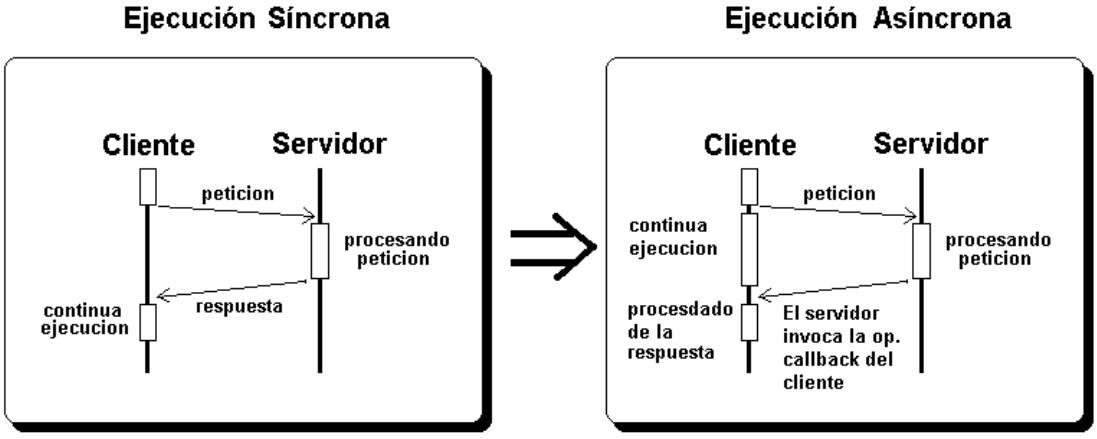


Figura 3. Ejecución síncrona frente a asíncrona.

En primer lugar, el proceso cliente invocará el método del proceso servidor que gestione su demanda, y posteriormente el proceso servidor invocará el método callback cuando termine de realizar la operación que el cliente solicitó.

3.2. Descripción del patrón.

3.2.1. Intención.

- La intención del patrón es permitir que se puedan realizar operaciones en el cliente mientras el servidor procesa la petición.

3.2.2. Motivación.

- Un cliente necesita el resultado de una operación remota, pero no puede permitirse suspender su ejecución en espera de la respuesta de dicha operación.

- La utilización del Servicio de Eventos CORBA no es eficiente (ver 3.2.4) en los casos en los que se trata de continuar la ejecución en el cliente mientras el servidor procesa la petición.

3.2.3. Aplicabilidad.

El patrón callback se podrá aplicar en aquellos sistemas en los que:

- El proceso cliente necesita ejecutar otras tareas en paralelo a la operación del proceso servidor que le proporciona el resultado.

- El cliente necesita tener varios procesos ejecutándose en paralelo, de forma que cada uno de ellos puede realizar una petición y recibir los resultados en cualquier orden.

- El cliente solo quiere aceptar el resultado de la petición si se le proporciona en un intervalo de tiempo determinado (time-out). El cliente debe soportar la opción de cancelar la petición, en cuyo caso tampoco se aceptará el resultado.

3.2.4. Consecuencias: ventajas e inconvenientes.

Ventajas:

- El cliente no tiene que esperar a que acabe el servidor ya que, al realizar el proceso cliente la petición, éste no cede el control al servidor suspendiendo su ejecución en espera de los resultados, sino que continúa ejecutándose. De esta forma, el proceso cliente puede recibir los resultados cuando el servidor los tenga disponibles sin tener que bloquearse mientras los obtiene.

- Al permitirse que la operación *callback* se pueda ejecutar de manera concurrente, el cliente puede manejar respuestas múltiples de varios servidores o realizar varias peticiones en el orden en que se produzcan.

- El cliente puede rechazar aquellas repuestas que ya no sean necesarias, bien por expirar el plazo de tiempo permitido o por cualquier otra causa (por ejemplo la cancelación de la operación). En el caso de que expire el tiempo establecido, la operación *callback* no procesará los datos o volverá a pedirlos, ahorrándose el tiempo perdido en este proceso. Si no se utilizara el patrón, el proceso cliente quedaría suspendido durante el tiempo de time-out y habría que realizar la petición de nuevo.

- A pesar de que el **servicio de eventos de CORBA** proporciona la misma funcionalidad en este problema concreto, la utilización del

patrón *callback* difiere del servicio de eventos en los siguientes puntos:

Desventajas del Servicio de Eventos de CORBA:

- Es necesario inicializar el servicio de eventos, lo cual es una tarea complicada. Esto es una ventaja del *callback* sobre el servicio de eventos, ya que en el primero no es necesaria ninguna inicialización (la inicialización del ORB es común a ambas soluciones).

- El servicio de eventos necesita un canal de eventos virtual sobre el ORB, al cual acceden tanto productores como consumidores (utilizando el modelo *push* o *pull* del servicio de eventos). El control del canal de eventos hace que aumente la sobrecarga de la comunicación aumentando la cantidad de información de control que hay que transmitir a través del ORB. En cambio, usando *callback* no es necesario crear este canal ya que la comunicación es directa a través del ORB. Las llamadas a los métodos remotos se realizan utilizando referencias a objetos remotos, con lo que se consigue reducir la complejidad de las comunicaciones y, por consiguiente, el tiempo empleado en ellas.

- A la hora de utilizar el Servicio de Eventos de CORBA se presentan serios problemas de compatibilidad que se deben a que cada fabricante interpreta la norma CORBA a su modo y desarrollan productos que en muchos casos son incompatibles entre sí. Incluso en algunos casos los fabricantes ni siquiera proporcionan una implementación del Servicio de Eventos.

Ventajas del Servicio de Eventos de CORBA:

- El Servicio de Eventos de CORBA proporciona una funcionalidad mucho más amplia que el *callback,* pudiéndose utilizar en multitud de casos en los que la solución propuesta en este documento no es aplicable. Una de las características del Servicio de Eventos a tener en cuenta en la comparación con el patrón callback es que desacopla por completo el cliente del servidor, cosa que el patrón no realiza por sí solo.

- Permite consultar el estado intermedio de una operación o recoger un resultado parcial sin implementar una nueva operación como seria necesario con el patrón callback.

Desventajas:

- El control de errores es deficiente debido a que no se pueden utilizar excepciones. Si se produce una excepción en el servidor, habrá que comunicárselo al cliente. Se puede optar por varias soluciones. La primera opción consiste en pasar un parámetro de tipo booleano que indique si la operación se realizó con éxito, y la segunda opción es implementar el método *callback* de manera que tome como parámetro una excepción de tipo genérico y que dentro de su código distinga de qué tipo de excepción se trata e implemente las acciones pertinentes.

- El cliente debe implementar un método para recoger la respuesta. Éste será el método al que el servidor llame cuando tenga el resultado que necesita el cliente.

- Se produce un acoplamiento entre el cliente y el servidor. La solución a este problema es pasar un cliente de tipo abstracto en cuya interfaz se encuentre el método abstracto *callback*, que será implementado de manera diferente por cada uno de los clientes concretos que heredan de la clase abstracta. Este problema y su solución se detallan en el capítulo 4.

3.2.5. Implementación.

Los pasos a seguir para realizar la implementación del patrón callback distribuido son los siguientes:

- El primer paso es definir las interfaces del cliente y el servidor en IDL.

- La operación de respuesta, que se llamará callback, deberá contener parámetros para toda la información que el servidor producirá.

- Para mejorar la eficiencia la operación callback del cliente será declarada como una operación de tipo *oneway* de CORBA. De este modo, al llamar a la operación el proceso llamante no queda bloqueado hasta la finalización de ésta, sino que continúa su ejecución.

- Cada operación síncrona del servidor se convertirá en una operación asíncrona de tipo *oneway* con un parámetro adicional para controlar el callback.

- Los parámetros de las operaciones del servidor cambiaran según el siguiente criterio:

    1. Los parámetros de **entrada** no cambian.

2. Los parámetros de **entrada/salida** pasaran a ser sólo de **entrada** y se incluirán en la operación callback del cliente.

3. Los parámetros de **salida** serán eliminados de la operación del servidor y serán incluidos en la operación callback del cliente.

4. Por último, si la operación devuelve algún tipo se remplazará por void, y se añadirá un parámetro en la operación callback del cliente.

Ejemplo:
Si se parte de esta interface:

```
interface Ejemplo {
   string obtenerInf(in string nombre,
                     inout int edad,
                     out int sueldo);
};
```

aplicando el patrón se obtendrá ésta otra:

```
forward interface EjemploCallback;

interface Ejemplo{
   oneway void obtenerInf(in string nombre,in int edad,
                          in EjemploCallback callback);
};
interface EjemploCallback{
   oneway void callback (in int edad,
                         in int valor_devuelto_sueldo,
                         in boolean sin_fallos);
};
```

4. EJEMPLO DE APLICACIÓN.

4.1. Introducción.

A continuación se propone un ejemplo práctico que utiliza el modelo de tres capas, usando CORBA [1][4] para realizar la comunicación entre capas. Se implementará una comunicación asíncrona mediante la utilización del patrón de diseño callback distribuído.

Se trata de una aplicación en la que un cliente accede a una base de datos remota. Este tipo de aplicaciones se suelen implementar mediante el modelo cliente/servidor, pero en este caso se ha realizado un diseño en tres capas para aprovechar las ventajas que éste ofrece.

En el ejemplo propuesto, dos clientes diferentes acceden a una base de datos que contiene información sobre los empleados de una empresa. El cliente tendrá la opción de introducir un nuevo empleado en la base de datos, eliminarlo de la misma o bien consultar sus datos.

En la aplicación original, cuando se realizaba una consulta de empleado, el proceso cliente permanecía bloqueado hasta recibir el resultado de la operación. El objetivo que se persigue con la aplicación del patrón callback distribuido es conseguir que el cliente pueda seguir realizando peticiones aunque no haya recibido las respuestas de éstas (se podrían, por ejemplo, realizar tres peticiones consecutivas de datos de tres empleados distintos).

4.2. Descripción de la aplicación a desarrollar.

La aplicación que se describe como ejemplo se corresponde con una arquitectura típica cliente-servidor, en la que un cliente accede a los datos de una base de datos que se encuentra en otra máquina. Una arquitectura cliente-servidor se podría representar mediante la siguiente figura:

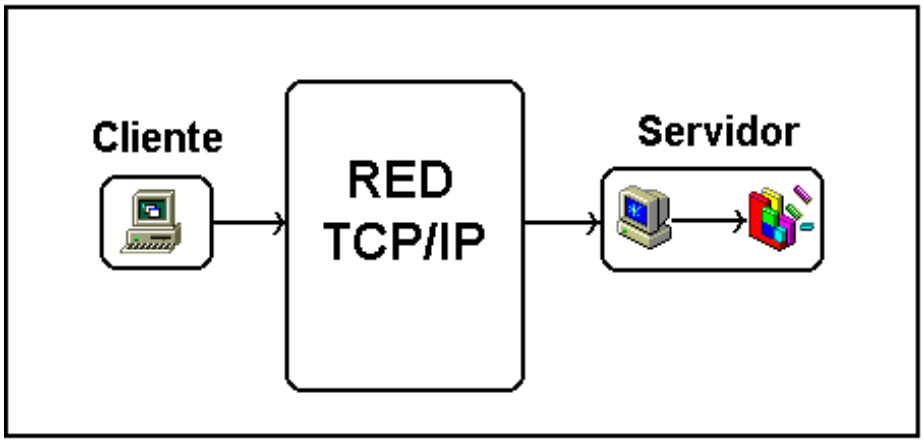


Figura 4. Arquitectura Cliente-Servidor.

Sobre este esquema se ha aplicado el modelo en tres capas estableciendo una capa para el Cliente, otra para el Agente (lógica de negociado) y una última para el Servidor de Bases de Datos. La figura siguiente representaría el nuevo esquema de la aplicación:

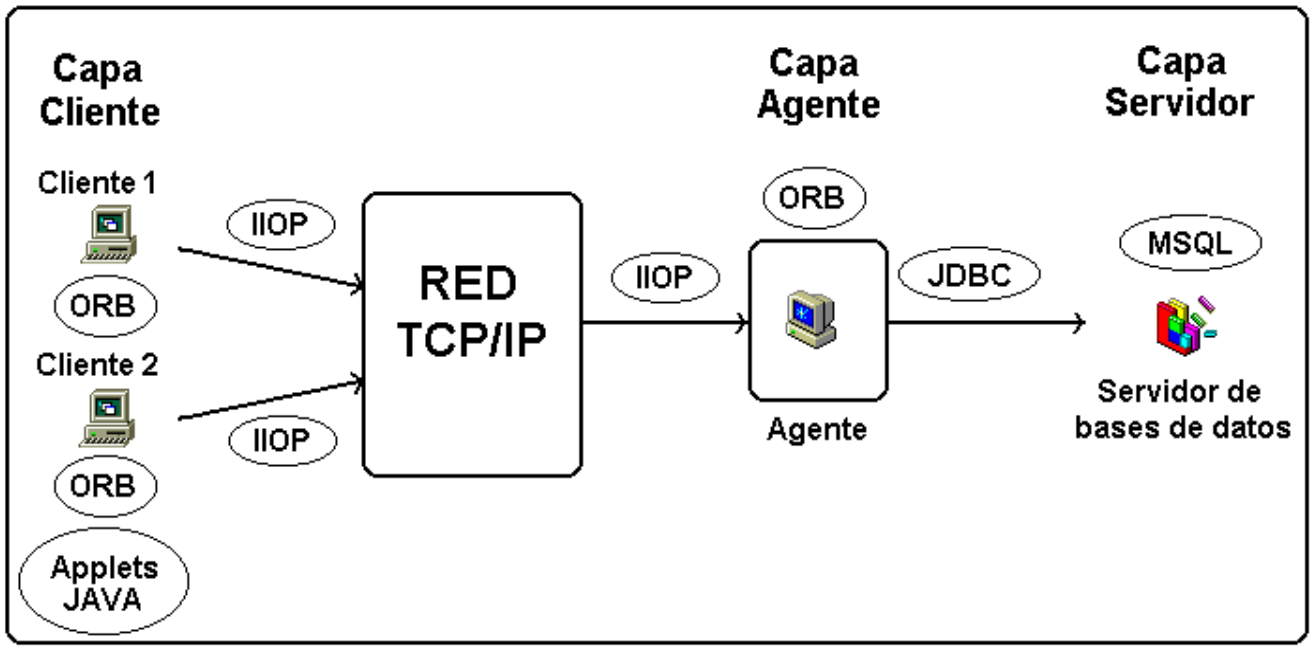


Figura 5. Modelo en tres capas aplicado a la aplicación.

Capa Cliente:

Como se observa en la figura anterior, se dispone de dos clientes que se comunican con el agente. Cada uno de ellos posee una interfaz de usuario

distinta, en la que se proporciona la funcionalidad necesaria para realizar las operaciones mencionadas anteriormente: introducir empleado, eliminar empleado y obtener datos de un empleado.

Si la petición del usuario es “introducir empleado” o “eliminar empleado”, el cliente se limita a enviar la petición a la capa agente, que ejecutará las acciones necesarias para atenderla.

Si la petición es “obtener empleado”, el cliente realiza la petición al agente, el cual ejecutará las acciones necesarias para devolverle los datos del empleado solicitados. Una vez el cliente tiene los datos, realiza un tratamiento sobre ellos; en este caso consistirá únicamente mostrarlos por pantalla.

- **Capa Agente:**

El agente se encarga de atender las peticiones del cliente. Cuando el cliente llame a una operación, ésta se solicitará al agente, que la procesará de forma adecuada. Las operaciones pueden ser de consulta contra la base de datos o de modificación de la misma.

- **Capa Servidor:**

Esta capa contiene la base de datos de los empleados, y atiende a las peticiones de la capa agente proporcionando los datos requeridos si se trata de la operación “obtener empleado”, o bien modificando la base de datos si la operación es “introducir empleado” o “eliminar empleado”.

Existen muchas configuraciones posibles a la hora de distribuir las capas del modelo en máquinas concretas, gracias a la versatilidad que proporciona el diseño utilizado. El sistema de tres capas propuesto está pensado para ejecutarse sobre varias máquinas localizadas en lugares diferentes.

Cada uno de los clientes se podrá ejecutar en una máquina. Para ejecutar el cliente es necesario únicamente un navegador que se conectará con el servidor HTTP y ejecutará el applet JAVA que contiene la aplicación cliente.

Este applet se comunicará con el agente utilizando CORBA. La comunicación se realizará entre el ORB del Cliente y el del Agente a través de Internet con el protocolo IIOP (Internet Inter ORB Protocol). El IIOP permite comunicar dos ORBs que se encuentren en redes distintas consiguiendo así un único ORB virtual.

La comunicación entre el Cliente y el Agente también podría realizarse a través de una LAN evitando la utilización del IIOP, ya que el ORB del cliente y el Agente estarían en el mismo dominio.

El Agente se encuentra en otra máquina y se comunica tanto con los clientes como con el Servidor de Bases de Datos. Para comunicarse con el

Servidor de Bases de Datos, el Agente utiliza JDBC, mediante el cual una aplicación JAVA puede realizar consultas en lenguaje SQL contra un servidor de Bases de Datos.

JDBC puede utilizarse con cualquier servidor que reconozca el lenguaje SQL, por lo que resulta más flexible que cualquier producto concreto que se utilice para acceder a una base de datos concreta. Únicamente es necesario utilizar el driver JDBC correspondiente a la base de datos a la que se quiera acceder. En el caso de las bases de datos de Microsoft, es necesario utilizar un driver JDBC que adapta las peticiones a ODBC (driver de acceso a las bases de datos de Microsoft), el cual realiza la consulta concreta a la base de datos.

En esta aplicación se ha utilizado el Servidor de Bases de Datos MSQL, presente en otra máquina diferente que recibe las consultas SQL del Agente y le proporciona los resultados de las mismas.

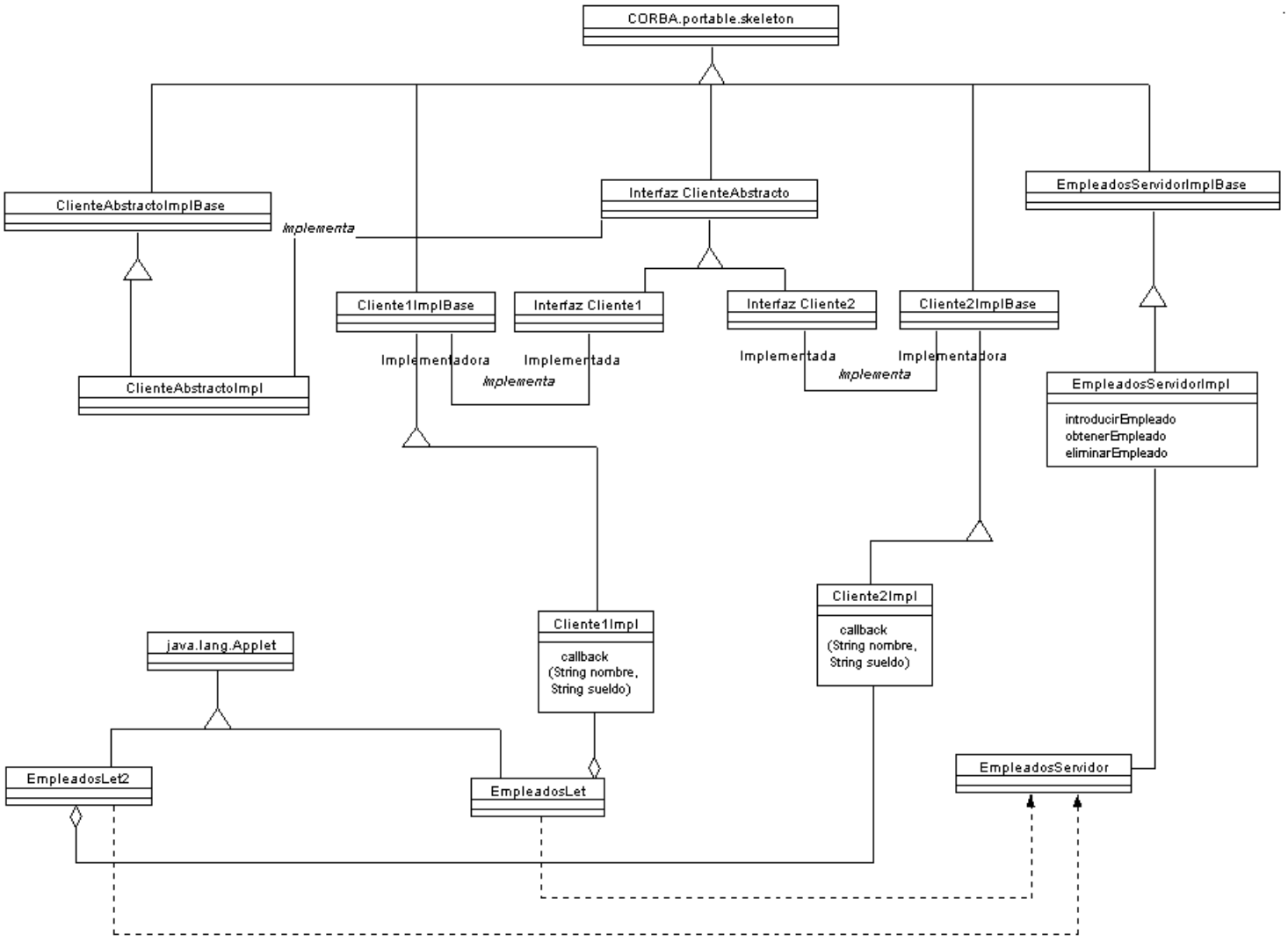
Es importante tener en cuenta que no es esta la única configuración posible a la hora de implementar la aplicación, existiendo una gran cantidad de variantes. Se podría, por ejemplo, tener múltiples clientes repartidos por todas las máquinas, así como varios Agentes para repartir la carga de clientes. Incluso el Servidor de Bases de Datos puede encontrarse en cualquier otra máquina, mientras se indique su dirección IP en el código de implementación del Agente (la modificación es trivial, ya que sólo es necesario modificar una línea de código).

Es posible, además, que tanto los clientes como el Agente estén implementados en cualquier lenguaje que soporte CORBA. El cliente únicamente debe respetar la interfaz que le proporciona el servidor. Si cambia la implementación del servidor, lo único que habrá que modificar será el acceso a la base de datos (JDBC es sólo para JAVA).

4.3. Modelado de la aplicación.

4.3.1. Modelo de clases.

Para facilitar la comprensión del diagrama de clases no se han representado todas las clases utilizadas. El conjunto de las clases no representadas son las que implementan algunas de las operaciones internas de CORBA. El modelo de clases de la aplicación se corresponde con el siguiente diagrama.



4.3.2. Descripción del modelo de clases.

Para la comprensión del modelo de clases es necesario tener presente que JAVA no permite herencia múltiple. Para emularla se utilizan *interfaces,* que se declaran de manera abstracta y contienen la signatura de un conjunto de operaciones que han de ser definidas por otra u otras clases.

- ***Clase EmpleadosServidorImpl:***

Esta es la clase que implementa la interfaz de *Empleados.* En ella reside la capa Agente, y será la encargada de proporcionar la interfaz que utilizará el cliente para realizar sus peticiones. Esta clase hereda de *EmpleadosServidorImplBase,* que se encarga de implementar funciones

internas de CORBA, que se usan para gestionar la comunicación a través del ORB (al igual que todas las clases que terminen en *ImplBase*, que son generadas automáticamente por el compilador de IDL).

Esta clase implementa las funciones:

- `public synchronized void` ***obtenerEmpleado***`(String rPrimerApellido, ClienteAbstracto mi_cliente)`
- `public synchronized void` ***introducirEmpleado*** `(String rPrimerApellido,String rNombre,String rSueldo)`
- `public synchronized void` ***eliminarEmpleado*** `(String rPrimerApellido)`

Cuando finalice su ejecución, la operación *obtenerEmpleado* llamará al método *callback* del objeto mi_cliente que se le pasa como parámetro pasándole los resultados para que ésta los procese (esto se explicará mas detalladamente en el punto 4.4).

- ***Clase EmpleadosServidor:***

Esta clase posee un objeto incrustado de tipo *EmpleadosServidorImpl* que implementa la agregación con la clase *EmpleadosServidorImpl.*

El objeto incrustado *EmpleadosServidorImpl* será dado de alta en el ORB una vez inicializado este, y servirá para implementar la agregación que se establece de manera virtual, a través del ORB, entre *EmpleadosServidor* y *EmpleadosLet* (que corresponde al cliente).

- ***Clases Cliente1Impl y Cliente2Impl:***

Estas dos clases son las encargadas de implementar los interfaces *Cliente1 y Cliente2* que heredarán de la interfaz *ClienteAbstracto.*

En ellas se encuentra únicamente la implementación de la operación

- `public void` ***callback***`(String cNombre, String cSueldo)`

cuya signatura es igual para ambas clases, siendo su implementación distinta. Esta operación se encarga de procesar la información recibida del Agente, lo que se realiza de manera diferente en cada cliente.

Las Clases Cliente1Impl y Cliente2Impl heredan de Cliente1ImplBase y Cliente2ImplBase.

- ***Clases EmpleadosLet1 y EmpleadosLet2:***

Su función es implementar la capa cliente. Se dispondrá de dos clientes diferentes con interfaces de usuario distintas. Es necesario que ambas clases hereden de la clase Applet, debido a que en JAVA todos los applets deben hacerlo para redefinir los métodos *init, run* y *paint*. *EmpleadosLet1* y *EmpleadosLet2* se encargan de inicializar el ORB y de obtener la referencia al objeto de tipo *EmpleadoServidorImpl*, que utilizarán como objeto incrustado virtual para poder llamar a través de éste a las operaciones que implementa el servidor (*obtenerEmpleado, introducirEmpleado y eliminarEmpleado*).

Gracias a la utilización de CORBA, el objeto *EmpleadosServidorImpl* se podrá usar como si se hubiera declarado de la forma habitual, reduciéndose la llamada a los procedimientos implementados por el servidor a:

```
- objetoServidorImplRemoto.obtenerEmpleado(...)
```

4.4. Utilización del patrón de diseño Callback Distribuído en la aplicación.

El objetivo de la utilización del patrón callback en esta aplicación es hacer que el cliente pueda seguir trabajando de manera normal mientras el servidor está procesando una petición de *obtenerEmpleado.* Como se puede observar en la siguiente traza de ejecución correspondiente a la aplicación original, el cliente quedaba bloqueado tras ejecutar la petición, esperando que el servidor respondiera con el resultado. Esto hace imposible introducir nuevos empleados o eliminarlos mientras se espera el resultado.

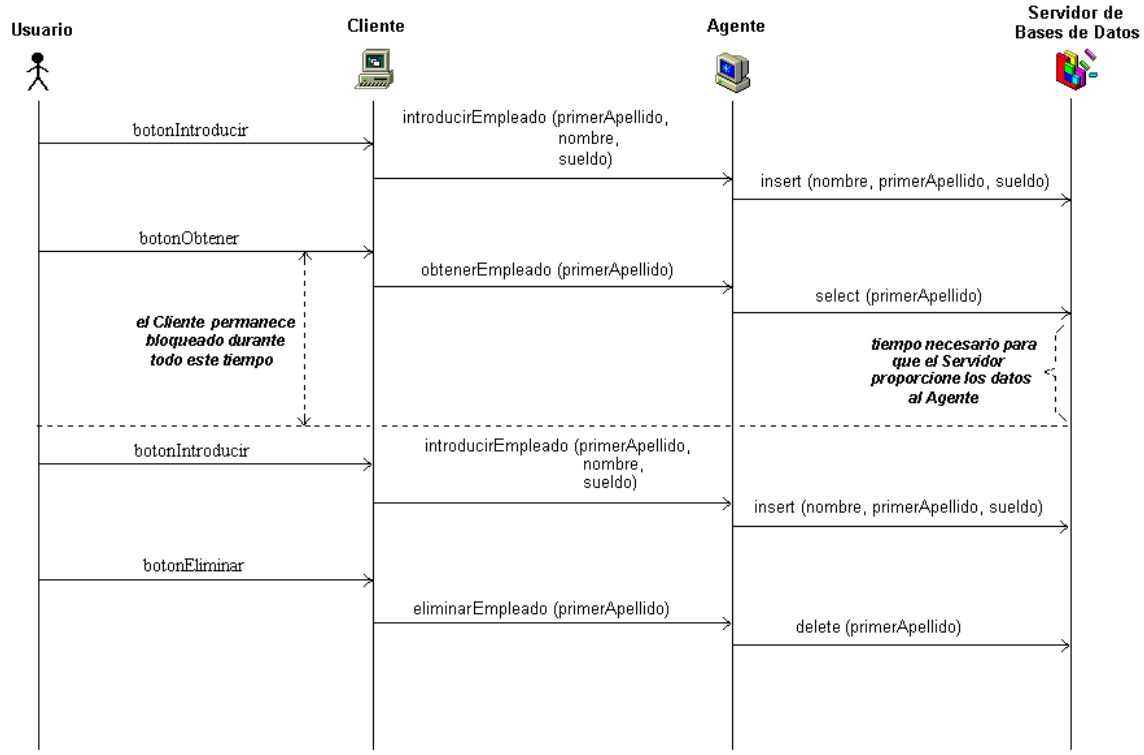


Figura 5. Traza de ejecución sin aplicar Callback.

Una vez introducido el patrón se consigue que, mientras se ejecute en el proceso servidor una petición de *obtenerEmpleado,* el cliente pueda seguir realizando peticiones. La nueva traza de ejecución sería la siguiente:

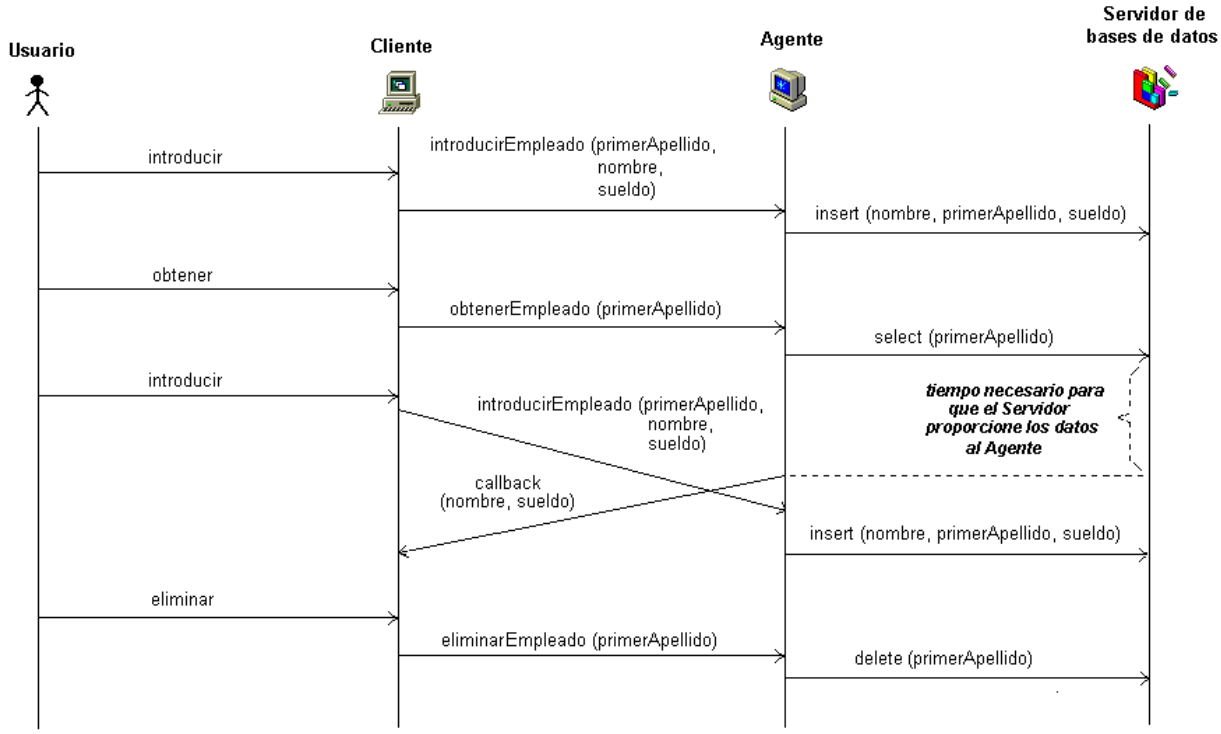


Figura 6. Traza de ejecución aplicando Callback.

En la figura se puede observar que, aunque el Agente admite múltiples peticiones del Cliente mientras se ejecuta en el Agente la operación *obtenerEmpleado*, las peticiones se procesan de manera secuencial en el mismo orden en que produjeron. Esto es debido a que las operaciones de acceso a la base de datos (*introducirEmpleado, eliminarEmpleado* y *obtenerEmpleado*) están declaradas como *synchronized*, lo que hace que no se puedan ejecutar de manera concurrente para mantener la consistencia de la base de datos.

Nótese que, a pesar de que no puedan realizarse múltiples accesos a la base de datos al mismo tiempo, el cliente sí puede realizar tantas peticiones como desee, ya que estas se encolarán a la espera de que finalicen las operaciones anteriores.

Para poder aplicar este patrón es necesario modificar el archivo IDL que contiene la especificación de la aplicación.:

- El IDL original de la aplicación era:

```
Module EmpleadosAp {
        interface Empleados {
                void obtenerEmpleado (in string rPrimerApellido,
                                      out string rNombre,
                                      out string rSueldo);
                void introducirEmpleado (in string rPrimerApellido,
                                         in string rNombre,
                                         in string rSueldo);
                void eliminarEmpleado (in string rPrimerApellido);
         };
 };
```

Una vez aplicamos el patrón callback con la mejora para desacoplar cliente y servidor (agente en el caso de tres capas) pasaría a ser:

```
Module EmpleadosAp{

   Interface ClienteAbstracto;

   Interface Empleados {

        oneway void obtenerEmpleado (in string rPrimerApellido,
                                     in ClienteAbstracto mi_cliente);

        void introducirEmpleado (in string rPrimerApellido,
                                 in string rNombre,
                                 in string rSueldo);

        void eliminarEmpleado (in string rPrimerApellido);
   };


   interface ClienteAbstracto{
             oneway void callback (in string cNombre,
                                   in string cSueldo);
   };

   interface Cliente1:ClienteAbstracto{};
   interface Cliente2:ClienteAbstracto{};
};
```

El procedimiento para transformar el antiguo IDL en el nuevo es el explicado en la sección 3.2.5. En este caso concreto, la única operación que requiere el uso del método callback es *obtenerEmpleado*, ya que solicita datos al servidor para que éste se los devuelva como parámetros de salida.

En la aplicación original sólo se disponía de un cliente. Ahora se han introducido dos clientes que poseen interfaces gráficas diferentes y procesan la

información recibida de manera distinta (implementaciones diferentes de la operación callback en cada uno de ellos).

Como ya se ha explicado, el principal problema del patrón callback es que provoca un acoplamiento entre el cliente y el servidor, ya que en la operación *obtenerEmpleado* se pasa como parámetro un objeto de tipo cliente para que el servidor pueda llamar a la operación callback de dicho cliente.

Debido a que se tienen dos clientes distintos, se debería implementar una operación obtenerEmpleado sobrecargada para cada uno de ellos. Por cada cliente diferente se debería realizar una nueva sobrecarga de la operación obtener Empleado.

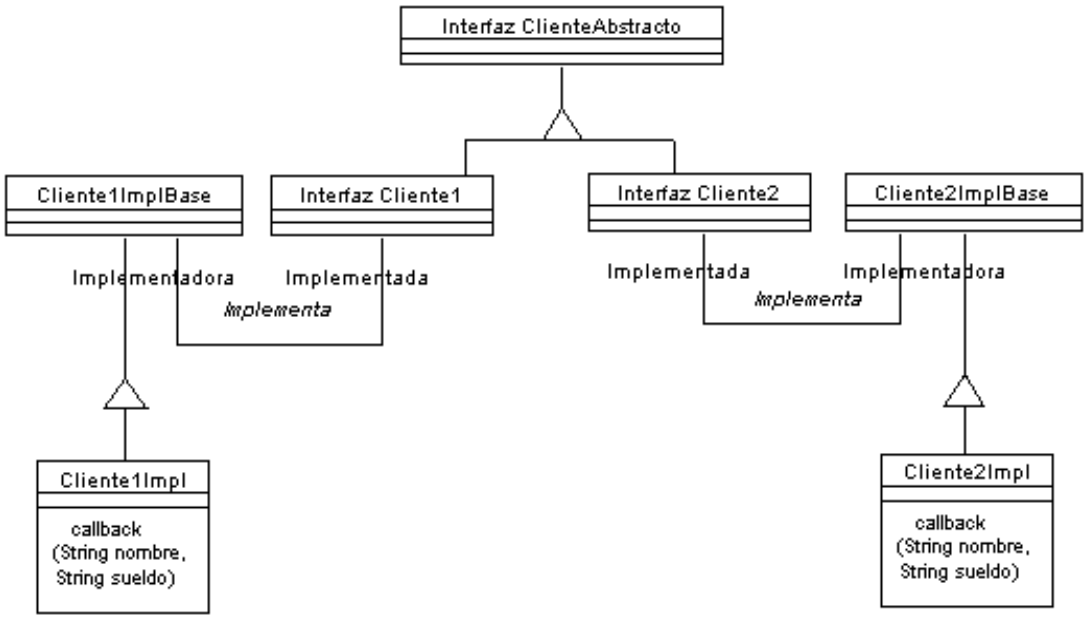
Existe también la posibilidad de recibir un objeto del tipo genérico Object (del que heredan todas las clases), y chequear el tipo original del objeto. Una vez se conoce el tipo se ejecutaría una sección de código determinada (con sentencias *if*, por ejemplo).

Tanto la primera como la segunda solución provocarían que el cliente y el servidor estuvieran fuertemente acoplados. Con el objetivo de evitar este acoplamiento y solucionar así el principal problema del patrón, se ha aplicado una variante del patrón comando.

La solución consiste en que el método *obtenerEmpleado* reciba como referencia al cliente un objeto de tipo *ClienteAbstracto*. La clase *ClienteAbstracto* es una clase de tipo abstracto en la que se define el prototipo de la operación callback. Todos los clientes concretos heredan de dicha clase, y cada uno de ellos implementa la operación callback de manera diferente, según las necesidades del cliente concreto.

De este modo es posible llamar a la operación *obtenerEmpleado* utilizando cualquier tipo de cliente, siempre que éste tenga implementada la operación callback. Así se pueden añadir todos los clientes concretos que se desee, con la única condición de que hereden de la clase *ClienteAbstracto* e implementen la operación callback.

Como se observa en la figura, las clases que implementan el callback son *Cliente1Impl* y *Cliente2Impl*. Estas clases implementan las interfaces *Cliente1* y *Cliente2*, que a su vez heredan de la interfaz abstracta *ClienteAbstracto*.

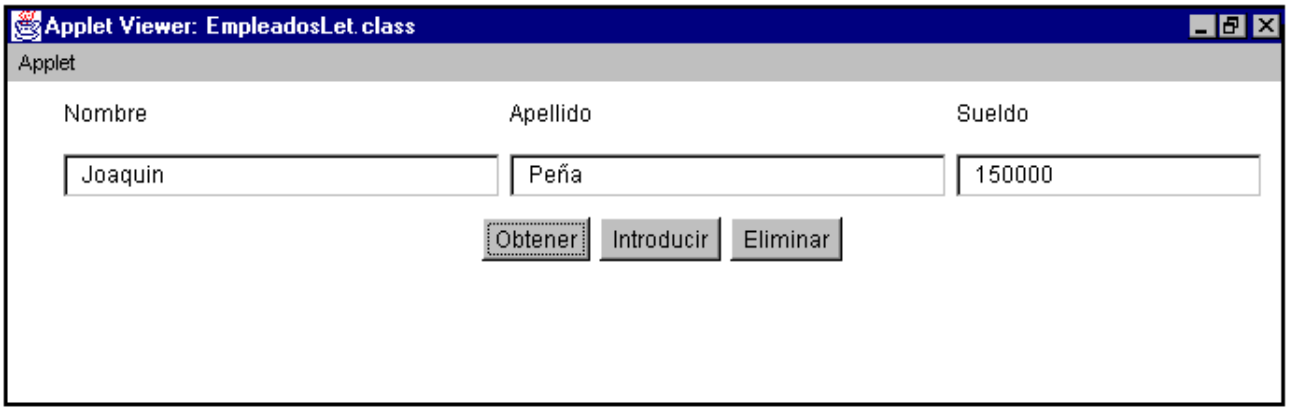

Interfaz ClienteAbstracto
Cliente1ImplBase
Interfaz Cliente1
Interfaz Cliente2
Cliente2ImplBase
Implementadora
Implementada
implementa
Implementada
Implementadora
implementa
Cliente1Impl
callback
(String nombre,
String sueldo)
Cliente2Impl
callback
(String nombre,
String sueldo)

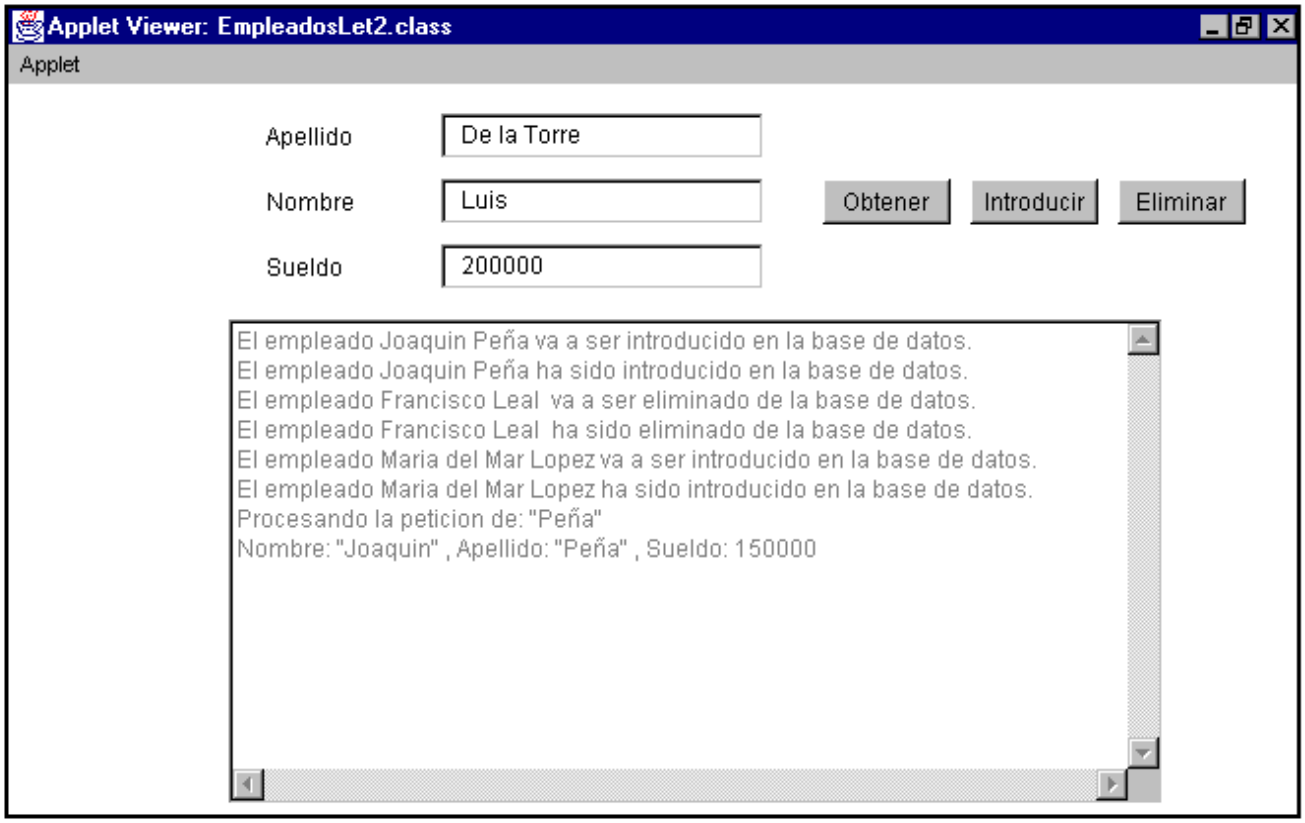
5. FUNCIONAMIENTO E INSTALACIÓN.

5.1. Funcionamiento.

Para comprender el funcionamiento de la aplicación, se deben tener en cuenta dos aspectos: la parte Cliente, con la que el usuario interactúa a través de las dos interfaces gráficas que se proporcionan y el Agente mostrará una serie de mensajes por pantalla detallando las actividades que está realizando.

Al ejecutar la aplicación, al usuario se le presenta la interfaz gráfica del cliente correspondiente. El Cliente Empleados 1 presenta la siguiente interfaz: mientras que el Cliente Empleados 2 posee ésta otra:





En ambas interfaces existen tres botones:

Introducir: se utiliza para introducir información sobre un nuevo empleado en la base de datos. Dicha información consiste en su nombre, su apellido y su sueldo. Para introducir un empleado en la base de datos es necesario proporcionar esta información rellenando los tres campos de texto correspondientes.

Eliminar: su función es eliminar un empleado de la base de datos. Es necesario introducir el apellido del empleado en el campo de texto *Apellido* para que éste pueda ser eliminado.

Obtener: mediante este botón se podrá obtener la información de un empleado que se encuentre en la base datos. Debe introducirse el apellido del empleado del cual se desea obtener información.

Una vez introducido el apellido y pulsado el botón, según el cliente que se esté ejecutando se realizarán las siguientes acciones:

- Cliente Empleados 1:

Transcurrirá un periodo de tiempo (el retardo ha sido introducido a propósito) hasta que la información esté disponible. En ese momento (y

si el empleado se encontraba en la base de datos), aparecerá la información del empleado en los campos Nombre y Sueldo.

- Cliente Empleados 2:

Aparecerá en el cuadro de texto inferior un mensaje indicando que la petición está siendo procesada.

Cuando la información del empleado esté disponible, ésta aparecerá en dicho cuadro de texto, tal y como se observa en la figura correspondiente a la interfaz de este cliente.

Todas estas acciones producen un efecto en el Agente, que mostrará mensajes indicando el estado en el que se encuentra y las acciones que está realizando:

Introducir: Aparecerá un mensaje por pantalla indicando los datos del empleado que se está introduciendo en la base de datos.

Eliminar: Aparecerá un mensaje por pantalla indicando el apellido del empleado que se va a eliminar.

Obtener: Aparecerá un mensaje de "Obteniendo datos para el empleado de apellido <apellido>". Una vez obtenida la información aparecerá un mensaje mostrando los resultados (nombre, apellido y sueldo). Por último se mostrará un mensaje indicando que se cede el control al cliente para que procese los datos.

6. PROBLEMAS Y CONCLUSIONES.

6.1. Problemas.

- Con respecto a la implementación del método *callback* se podría pensar que la solución más sencilla es implementar el método callback en las clases EmpleadosLet y EmpleadosLet2 y que ambos hereden de un cliente abstracto. Esto no es posible debido a la imposibilidad de utilizar herencia múltiple con JAVA, ya que forzosamente ambos clientes deben heredar de la clase Applet, no pudiendo heredar de la clase cliente abstracto necesaria para desacoplar cliente y servidor.

Para evitar el problema de la herencia múltiple, se podrían declarar interfaces que fueran implementadas por los clientes concretos, pero esto tampoco es posible porque deberían heredar de las clases de tipo *ImplBase* (funciones internas de CORBA), y las interfaces no pueden heredar de clases.

Así pues, la solución propuesta parece ser la más factible respecto a nuestros conocimientos de JAVA

- En principio, puede parecer poco ortodoxo que las clases *ClienteImpl1* y *CleinteImpl2* posean como atributos los TextArea de la interfaz gráfica, pero esta ha sido la única solución que se ha encontrado para que el método callback pudiera modificar el contenido de éste teniendo en cuentan que, como se acaba de explicar, no es posible introducir el método callback en la clase *EmpleadosLet*.

- Otro de los problemas surge a la hora de intentar ejecutar los applets clientes utilizando Internet Explorer. Con este navegador se producen incompatibilidades debido a que implementa la mquina virtual de microsoft, diferente al estándar especificado por Sun.

6.2. Conclusiones.

Se ha conseguido una aplicación considerablemente flexible y portable que posee, además, un elevado grado de escalabilidad.

A pesar de la introducción del patrón callback se han mantenido desacoplados el cliente y el servidor.

Gracias a la realización de este documento se han adquirido conocimientos para el desarrollo de aplicaciones en tres capas, aplicaciones CORBA y en la aplicación práctica de un patrón de diseño.

Al aplicar el patrón callback ha sido posible conocer los problemas derivados de la utilización de un patrón de diseño en una aplicación real, entre

los cuales cabría destacar la imposibilidad de aplicar el patrón en su forma original, además de la necesidad de combinarlo con otros patrones para conseguir la funcionalidad deseada.

7. BIBLIOGRAFÍA

[1] Client/Server Programing with JAVA and CORBA. Second Edition Robert Orfali, Dan Harkey. Editorial Wiley 1998

[2] CORBA Design Pattern, Thomas J. Mowbray, Raphael C. Malveau, Wiley, 1998.

[3] Erich Gamma et al. "Design Patterns". Addison-Wesley. 1994

[4] La Biblia de JAVA. Editorial Anaya.