UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Servicio de eventos. Comunicación asíncrona entre componentes de aplicaciones distribuidas

TESIS

Que para obtener el título de

Ingeniero en Computación

P R E S E N T A

Liliana Gutiérrez Flores

DIRECTOR DE TESIS

Ing. Manuel Manríquez Miranda

Ciudad Universitaria, Cd. Mx., 2003

CONTENIDO

LISTA DE FIGURAS Y TABLAS

página

INTRODUCCIÓN

El objetivo del presente trabajo es presentar un sistema que resuelva las necesidades de comunicación entre las aplicaciones de una empresa y que permita la automatización de los procesos de negocio de la misma. El sistema permite integrar las aplicaciones de una empresa en forma estándar, flexible y dinámica. Es estándar por dos motivos: primero, porque está basado en un estándar de industria; y segundo, porque incluye todas las formas de comunicación que se pueden dar entre las aplicaciones de una empresa, es decir que puede adoptarse como la forma estándar de comunicación entre las aplicaciones de la empresa. Es flexible porque permite integrar diferentes tipos de aplicaciones independientemente de la tecnología de desarrollo, de los sistemas operativos y de la arquitectura de los equipos. Y finalmente es dinámica porque permite hacer cambios en las aplicaciones sin impactar al resto.

La forma de comunicar aplicaciones en la actualidad se conoce como punto a punto. En una comunicación punto a punto, las dos partes interesadas se ponen de acuerdo en la información que van a compartir y en la forma de comunicación, esto ocasiona que las aplicaciones se queden de cierta forma atadas. Cuando una de las aplicaciones tenga que evolucionar ya sea funcional y tecnológicamente, deberá tener especial cuidado en el impacto que esto ocasionará en la otra aplicación.

La principal problemática cuando se tiene una gran cantidad de interfaces punto a punto, es el alto costo del mantenimiento de las mismas. Un pequeño cambio en una aplicación que ha acumulado un gran número de interfases, puede impactar a gran número de las aplicaciones con las que interactúa.

En la mayoría de las empresas existen aplicaciones a las que se denomina “heredadas”, estas aplicaciones diseñadas para algún propósito específico se han ido perfeccionando y adecuando a la empresa con el paso del tiempo. Típicamente las aplicaciones heredadas están desarrolladas con tecnologías obsoletas y son ejecutadas en equipos de arquitecturas antiguas, sin embargo es difícil cambiarlas por aplicaciones nuevas debido a la experiencia de la empresa que hay detrás de ellas. Otra problemática asociada a las aplicaciones heredadas, es que las nuevas tecnologías de comunicación de aplicaciones, no aplican para equipos tan antiguos, por lo que además es difícil comunicarse con ellas.

Para lograr la automatización de un proceso de negocio de una empresa, se deben integrar aplicaciones nuevas con aplicaciones heredadas. Podemos pensar que la aplicación que dará soporte a un proceso particular, es una aplicación que tiene su funcionalidad distribuida en las diferentes aplicaciones que apoyan el proceso. La problemática ahora es como hacer que las aplicaciones en un ambiente heterogéneo se comuniquen sin pagar los altos costos asociados a las interfaces punto a punto.

En el primer capítulo se abordan de forma breve diferentes tópicos asociados a los ambientes de cómputo. En los antecedentes hablamos de los hechos históricos que han determinado la forma de desarrollar aplicaciones en la actualidad, posteriormente repasamos el modelo cliente – servidor y los diferentes esquemas de comunicación entre aplicaciones, a continuación hablamos de la forma en que se han desarrollado las aplicaciones desde los

inicios de la computadora hasta nuestros días y finalmente se plantea de una manera formal la problemática a resolver con el presente trabajo.

En el segundo capítulo hablamos de las diferentes formas de comunicar aplicaciones que se han utilizado a lo largo del tiempo. Primeramente hablamos de la forma de intercambiar datos entre sistemas, posteriormente de la integración a nivel aplicativo y finalmente de herramientas de software de conectividad y de servicios distribuidos.

En el tercer capítulo se habla de la arquitectura de comunicación en un esquema de eventos y de los estándares de eventos, notificación y nombres de la OMG. La OMG (Object Management Group) es el organismo que mantiene el estándar CORBA incluyendo los diferentes servicios para aplicaciones distribuidas entre los cuales está el servicio de eventos. CORBA (Common Object Request Broker Architecture) es una Arquitectura que especifica la forma de comunicar los componentes que integran una aplicación.

En el cuarto capítulo se describe una propuesta para el Servicio de Eventos que cubre las necesidades de comunicación entre las aplicaciones de una empresa. Iniciamos con los requerimientos funcionales que incluyen la comunicación sincronía para la solicitud de servicios y el envío de mensajes de forma asíncrona a una o más aplicaciones. En los requerimientos técnicos se describen las características no-funcionales indispensables tales como la entrega garantizada de mensajes y la auditabilidad. A continuación se presenta el diseño del componente de software donde se describen las clases a utilizar y la secuencia de mensajes entre los objetos para las diferentes formas de comunicación. Finalmente se revisan las alternativas de lenguajes de programación para el desarrollo y se presenta una propuesta. Para este capítulo se utilizó el UML (Lenguaje Unificado de Modelado) que nos sirve para crear diagramas tanto para los requerimientos como para el diseño.

En el quinto capítulo hablamos de la implantación del servicio de eventos. Iniciamos con un ejemplo de las utilidad del Servicio de Eventos, la implantación de la funcionalidad y las especificaciones técnicas asociadas a los requerimientos técnicos presentados en el capítulo anterior.

I LOS AMBIENTES DE CÓMPUTO

I.I Antecedentes

Desde el inicio de la era de las computadoras hasta la década de los 80´s, las computadoras se caracterizaban por ser grandes y caras. En la mayoría de las organizaciones se contaba tan sólo con pocas computadoras y no había forma de conectarlas. Dominaban los sistemas centralizados con procesamiento paralelo y unas cuantas terminales.

Un sistema centralizado (*figura 1.1*) se compone de uno o más procesadores, memoria y dispositivos de almacenamiento. Una característica peculiar de un sistema centralizado es que todos sus procesos comparten memoria.

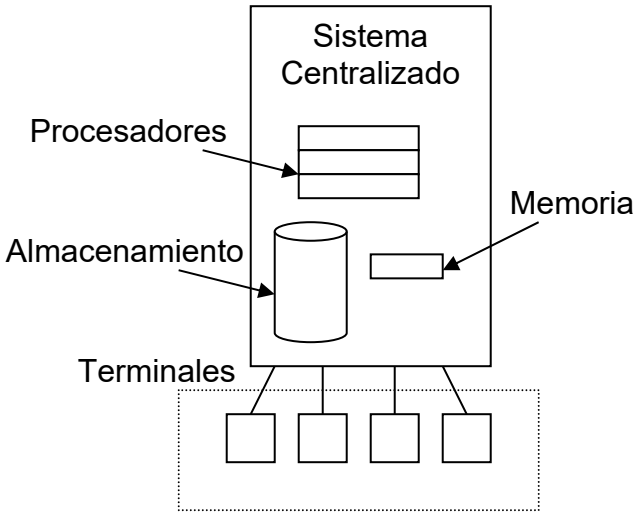


Figura 1.1. Diagrama de un sistema centralizado.

Las decisiones con relación a los equipos de cómputo en una organización están basadas en la proporción precio / rendimiento, en aquellas épocas prevalecía le ley de Grosh: “el poder de cómputo de un equipo es proporcional al cuadrado de su precio”, es decir que, por el doble del precio, se puede obtener un equipo con un poder de procesamiento cuatro veces superior al original. Por esta razón las organizaciones buscaban el equipo más grande que pudieran conseguir.

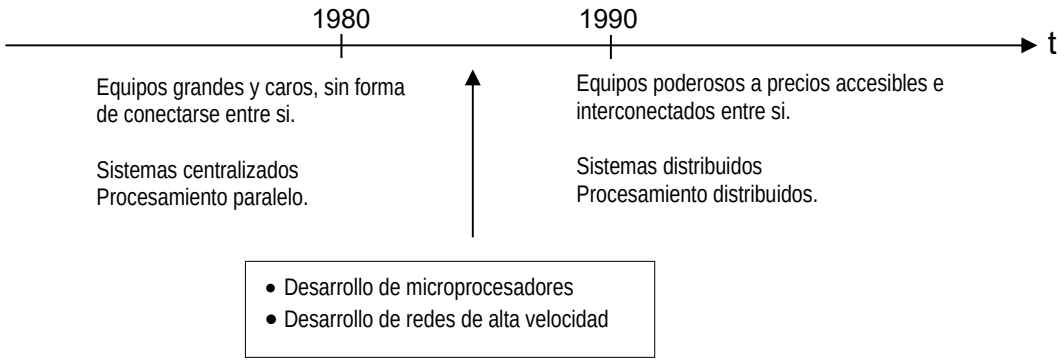


Figura 1.2. Evolución de los sistemas de cómputo.

Dos acontecimientos han cambiado la tendencia de los sistemas centralizados a los sistemas distribuidos: Las invención de los microprocesadores y el desarrollo de las redes de alta velocidad (*figura 1.2*). Con la invención de los microprocesadores, surgen equipos de cómputo con el poder de procesamiento de los grandes equipos centralizados por una fracción de su precio. La Ley de Ghosh ya no aplica más, ahora por el doble del precio, obtengo un equipo tan sólo un poco más rápido. Con el desarrollo de las redes de alta velocidad, resulta sencillo conectar varios equipos pequeños entre si y obtener una mejor proporción precio / rendimiento que con un sólo equipo grande.

Un sistema distribuido es un conjunto de equipos interconectados y trabajando de forma conjunta, a diferencia de los sistemas centralizados, los procesos pueden estar corriendo en diferentes espacios de direcciones. El modelo de comunicación mas utilizado en un sistema distribuido es el modelo cliente – servidor que se describe a continuación.

I.II El modelo cliente – servidor

Los sistemas se comunican entre si utilizando protocolos que especifican el formato y secuencia de los mensajes. Para facilitar el trabajo con los distintos aspectos relacionados con la comunicación, la ISO (Organización Internacional de Estándares) desarrollo el modelo de referencia para la interconexión de sistemas abiertos. *(Ver el apéndice D para detalle de las capas del modelo ISO-OSI).*

En el modelo cliente – servidor existe un grupo de procesos llamados servidores que ofrecen servicios a los clientes. Podemos encontrar ejemplos de comunicación utilizando el modelo – cliente servidor en una red local UNIX de estaciones de trabajo para la transferencia de archivos (ftp), para accesos remotos (telnet), para acceder a sistemas de archivos remotos (NFS), etc. Para la comunicación entre el cliente y el servidor se pueden utilizar dos clases de protocolos, sin conexión y con conexión. Cuando se utiliza un protocolo sin conexión, el cliente escribe su solicitud y anexa datos de identificación (dirección de red), a continuación el servidor envía su respuesta utilizando los datos de identificación recibidos (*figura 1.3*).

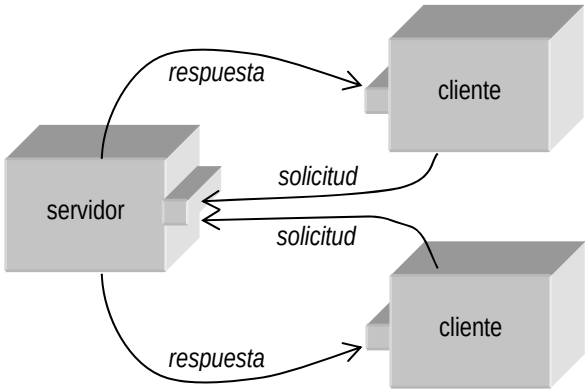


Figura 1.3. Protocolo sin conexión.

Cuando se utiliza un protocolo con conexión, la solicitud inicial del cliente sirve para establecer un canal de comunicación bi-direccional privado, de esta manera el cliente y el servidor pueden seguir interactuando sin necesidad de intercambiar un identificador de proceso en cada mensaje (*figura 1.4*).

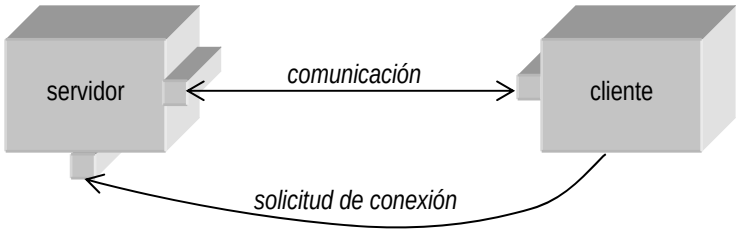


Figura 1.4. Protocolo con conexión.

Cuando un servidor recibe una solicitud de un cliente, se dedica íntegramente a atender esa solicitud y una vez que termina, atiende solicitudes adicionales. Para evitar largas esperas se utiliza la estrategia del servidor padre. Cuando el servidor recibe una solicitud, la bifurca a un hijo y vuelve a quedar en espera de solicitudes, de esta manera se pueden recibir múltiples solicitudes de manera sucesiva (*figura 1.5*).

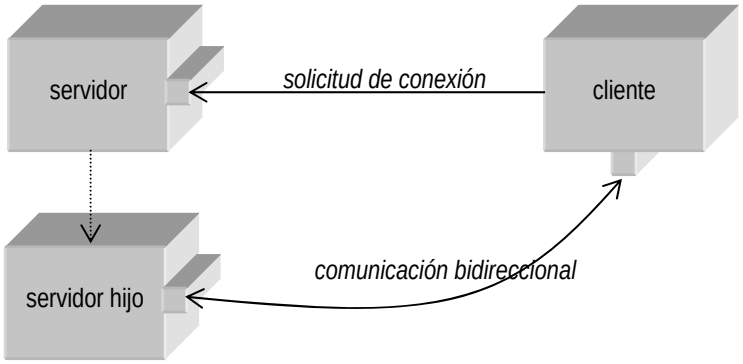


Figura 1.5. Estrategia del servidor padre.

Otra alternativa para evitar las esperas, es el manejo de hilos de ejecución. En esta estrategia el servidor crea un nuevo hilo de ejecución en su mismo espacio de proceso (*figura 1.6*). Este enfoque es útil cuando en las solicitudes no hay muchos cálculos, por ejemplo que predominen las operaciones de entrada - salida.

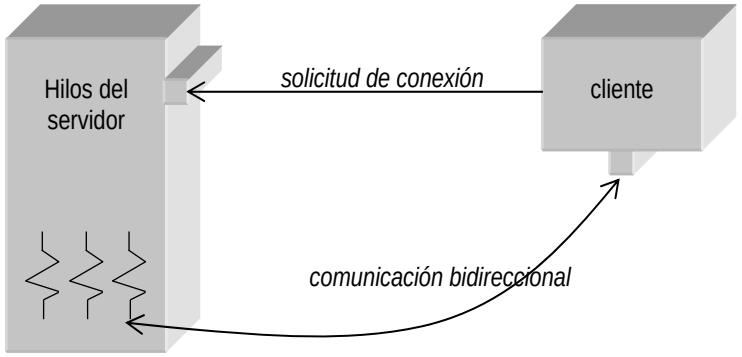


Figura 1.6. Manejo de hilos de ejecución.

I.III Esquemas de comunicación

Existen diferentes esquemas o paradigmas de comunicación entre sistemas cada uno apropiado para diferentes escenarios.

- Esquema de comunicación síncrono (solicitud / respuesta)
- Esquema de comunicación síncrono-diferido (manejo de colas de espera)
- Esquema de comunicación asíncrono (publicación / suscripción)

Esquema de comunicación síncrono

En el esquema de comunicación síncrono, un cliente hace una solicitud a un servidor, se queda en espera de la respuesta y una vez que la recibe, continua la ejecución de la aplicación. Este es el esquema de comunicación típicamente utilizado en el modelo cliente – servidor y se le conoce como el paradigma solicitud – respuesta.

Solicitud - respuesta

En el paradigma solicitud – respuesta, un cliente hace una solicitud a un servidor que lleva a cabo las acciones necesarias y entrega una respuesta. Si el servidor requiere información adicional para llevar a cabo la operación solicitada, esta debe ser enviada en forma de parámetros en la solicitud (*figura 1.7*).

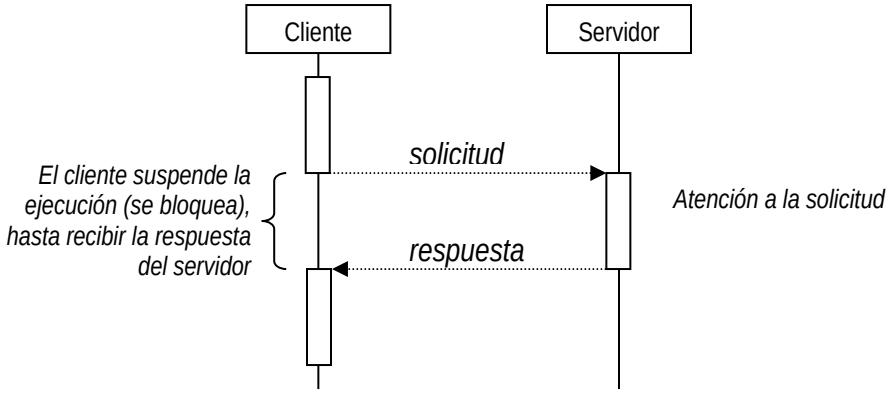


Figura 1.7. Solicitud - respuesta.

Este tipo de comunicación es adecuado cuando el cliente necesita la respuesta a la solicitud para continuar con la ejecución del programa.

Considerando un ambiente distribuido donde puede haber varios puntos de falla, ya sea el medio de comunicación o el sistema remoto, es posible que un cliente se quede esperando una respuesta que no llegará. Para evitar este problema se debe hacer un adecuado manejo de las excepciones que pueden ocurrir, incluyendo el manejo de tiempos de espera.

Esquema de comunicación síncrono diferido

El esquema de comunicación síncrono diferido propone el manejo de colas de espera para almacenar los mensajes que no puedan ser entregados de inmediato a su destino. Se pueden manejar colas tanto para almacenar solicitudes, como para almacenar respuestas.

Manejo de colas de solicitudes

El manejo de colas de espera propone el uso de algún tipo de almacenamiento persistente para almacenar las solicitudes hechas a un servidor, de esta manera cuando el servidor no está disponible o está ocupado, las solicitudes no se pierden y pueden ser procesadas cuando el servidor se restablezca. Una vez que las solicitudes se encuentran en una cola, se

pueden manejar políticas para asignar prioridades a las peticiones en la cola de espera (*figura 1.8*).

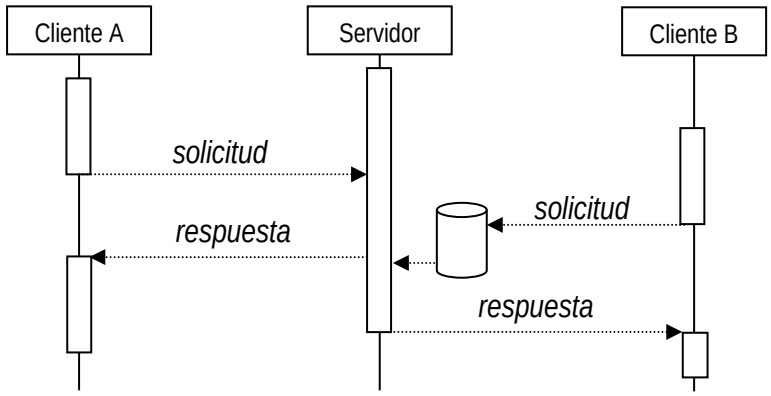


Figura 1.8. Colas de solicitudes.

Manejo de colas de respuestas

También se puede utilizar algún tipo de almacenamiento persistente para depositar la respuesta a una solicitud, el cliente deberá revisar el dispositivo de almacenamiento acordado, periódicamente, para verificar si ya está la respuesta a la solicitud (*figura 1.9*).

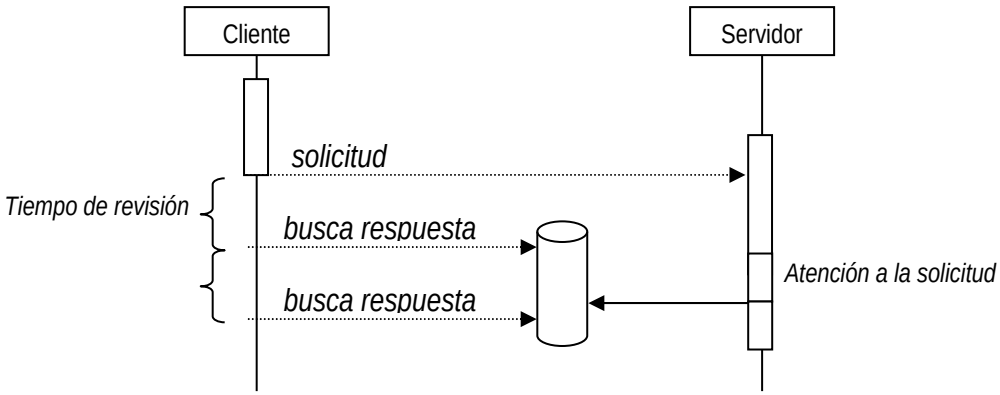


Figura 1.9. Colas de respuestas.

Esquema de comunicación asíncrono

Cuando el cliente hace una solicitud y no requiere la respuesta para continuar con la ejecución de la aplicación, puede hacer una solicitud asíncrona. En una solicitud asíncrona, el cliente no se queda en espera de la respuesta y no bloquea la ejecución de la aplicación. Cuando el servidor esta listo para entregar la respuesta, deberá informar a la aplicación

cliente para que se prepare para recibirla. Otra forma de comunicación asíncrona en el manejo de eventos con el paradigma publicación – suscripción.

Publicación - suscripción

En este esquema de comunicación, a diferencia del esquema solicitud - respuesta, el cliente no hace una solicitud de servicio, sino que hace una solicitud para ser notificado cuando ocurre algún evento en cierta aplicación (*figura 1.10*).

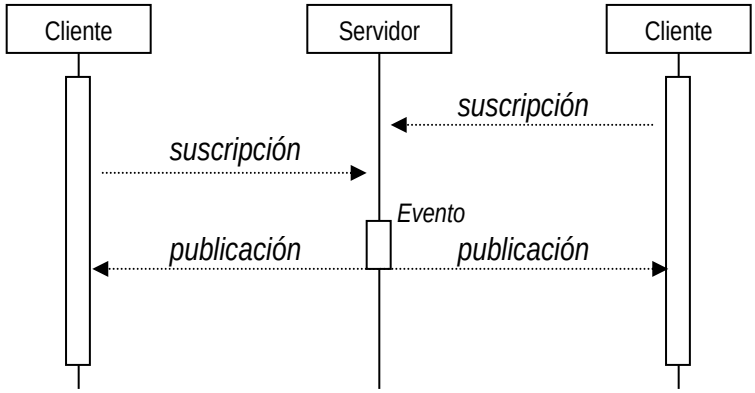


Figura 1.10. Publicación - suscripción.

Mediante el uso de eventos, una aplicación interesada el algún suceso en una aplicación remota, no tiene que estar preguntando periódicamente para verificar si ha ocurrido. La aplicación generadora del evento debe informar a todas las aplicaciones interesadas. El cliente o receptor del evento es interrumpido en su trabajo cuando ocurre un evento al que esta suscrito, muy seguramente debe realizar algo como consecuencia de la notificación de evento recibido. La principal complicación del manejo de eventos es que los generadores deben saber quienes son sus suscriptores y deben saber como hacerles llegar la notificación del evento.

I.IV Desarrollo de aplicaciones

Los primeros programas se realizaron con instrucciones en lenguaje máquina, en binario; posteriormente se utilizó el lenguaje ensamblador, que permite a los programadores utilizar una representación simbólica de las instrucciones y finalmente surgen los lenguajes de alto nivel con instrucciones más complejas que provocan la ejecución de más de una instrucción de lenguaje máquina.

Las computadoras ejecutan el lenguaje máquina que fue diseñado para el hardware específico (es dependiente del hardware). El lenguaje máquina se compone de cadenas de números, que son reducidos a 1’s y 0´s, e indican a la computadora que realice operaciones elementales una a la vez. Los lenguajes máquina son engorrosos para los seres humanos.

En los lenguajes ensambladores, se utilizan palabras y abreviaciones en ingles para representar las operaciones elementales de una computadora. Se utilizan ensambladores o traductores para convertir los programas a lenguaje de máquina que puedan ser ejecutados por las computadoras.

Los lenguajes de alto nivel pueden ser interpretados o compilados. Un lenguaje interpretado de alto nivel es traducido en tiempo de ejecución, para ello se utiliza un interprete que traduce cada una de las instrucciones a lenguaje máquina para que puedan ser ejecutadas. Los lenguajes interpretados son populares en ambientes donde se agregan nuevas características o se corrigen errores con frecuencia. Los lenguajes compilados son traducidos antes de su ejecución, se utiliza un compilador para generar código ejecutable para un hardware específico. Puesto que el código generado a partir del compilador será ejecutado sin tener que realizar ninguna tarea de interpretación, los programas escritos en lenguajes compilados son más rápidos en tiempo de ejecución.

Los primeros lenguajes de programación como BASIC o COBOL tienen la característica de ser no-estructurados. En un lenguaje no-estructurado el código no está organizado, todo el programa está en un sólo bloque. Cuando los desarrollos son muy grandes, la tarea de modificar el código o encontrar errores es muy complicada.

Los lenguajes de programación como Pascal o C se denominan lenguajes estructurados, estos lenguajes permite el manejo de bloques y subrutinas que permiten seccionar el código y mantener variables locales-temporales, esto hace que los programas sean mas sencillos de leer y mantener. Posteriormente aparece la programación orientada a objetos que ofrece una forma más intuitiva de conceptuar los problemas. A continuación se aborda con más detalle la programación estructurada y la programación orientada a objetos.

Programación estructurada

En los sesenta nace la programación estructurada que introduce la capacidad de seccionar el código de un programa. Entre los lenguajes estructurados más conocidos, podemos mencionar C, Pascal, Ada y Modula 2.

Un lenguaje estructurado permite el uso de bloques de código que son un conjunto de instrucciones relacionadas y que son tratados como unidad (en lenguaje C por ejemplo, los delimitadores de bloques son las llaves: {}). Los bloques pueden estar contenidos en construcciones de bucle tales como `while`, `do-while` y `for` o pueden estar dentro de una sentencia condicional `if`. Un lenguaje estructurado permite sangrar el código de un programa para facilitar su lectura y mantenimiento.

Se pueden crear subrutinas que realicen cierta tarea y que utilicen variables locales-temporales. Para utilizar dichas subrutinas sólo es necesario saber que tarea realizan y las variables de entrada y salida, no es necesario conocer como la realizan. Las subrutinas permiten esconder del resto del programa la información e instrucciones necesarias para realizar cierta tarea. Cuando una aplicación es desarrollada por un grupo de programadores, cada uno de ellos se enfoca en ciertas subrutinas sin tener que trabajar todos en el mismo archivo de código.

Programación orientada a objetos

En la programación orientada a objetos se utilizan objetos para implementar piezas de software. un objeto contiene métodos para implementar su comportamiento y variables para almacenar su estado. Se puede controlar el acceso a las variables permitiendo acceso a ellas sólo a través de los métodos diseñados para ello, a esto se le conoce como encapsulamiento.

Mediante el manejo de objetos obtenemos el beneficio de la modularidad, ya que el código de un objeto se mantiene independiente al resto de los objetos. Los objetos tienen una interfaz pública que es utilizada por otros objetos para comunicarse con ellos. Se puede mantener información propia del objeto en privado que puede ser modificada en cualquier momento sin afectar a otros objetos que se comunican con él.

Un objeto generalmente es un componente de una aplicación que se conforma de varios objetos. Los objetos se comunican entre sí mediante el envío de mensajes. Un mensaje consta de tres partes esenciales: el objeto destino, el método a ejecutar y los parámetros. Como podemos ver la comunicación entre objetos es a través de los métodos que definen su comportamiento y no accediendo directamente a las variables del objeto.

Una clase es un prototipo que define variables y métodos que son comunes a los objetos de cierto tipo. Una variable de clase contiene información que es compartida por todas las instancias(objetos) de esta clase. Si una instancia cambia una variable de clase, el cambio afecta también al resto de las instancias de la clase.

Las clases se organizan en forma de árbol, una clase de clases es llamada superclase y las clases que la conforman se conocen como subclases. Las subclases heredan las variables y métodos de clase de la superclase, pueden incorporar nuevas variables o métodos e incluso sobrescribir los que heredan. Se utilizan interfaces para definir los métodos de una clase sin revelar como son implantados.

C++ (1970) incorpora a C la habilidad de utilizar la tecnología orientada a objetos. C++ se mantiene como un lenguaje compilado con la eficiencia que esto representa.

Java (1995) introduce el concepto de Maquina Virtual. Las aplicaciones desarrolladas en Java al ser compiladas, no generan código máquina para algún hardware específico, generan código intermedio que es ejecutado por la Maquina Virtual Java. Un equipo que implemente la Máquina Virtual es capaz de interpretar el código generado por la compilación para ser ejecutado en el hardware específico. Con este concepto Java se convierte en un lenguaje multiplataforma.

El desarrollo de aplicaciones con la tecnología orientada a objetos no se limita a aplicaciones en un sólo equipo, los objetos pueden residir en equipos remotos. ‘Common Object Request Broker Architecture’ es una referencia para el desarrollo de aplicaciones de objetos distribuidos propuesta por la OMG. CORBA permite invocar objetos en cualquier parte de la red como si fueran objetos locales. La comunicación entre los objetos se realiza mediante ORB´s (Object Request Broker) que permiten al cliente invocar un objeto independientemente del lenguaje en que este desarrollado, el sistema operativo donde este

corriendo o su localización en la red. CORBA define también un lenguaje para la definición de interfaces 'IDL' *(Ver el apéndice D para detalle de las capas del modelo ISO-OSI).*

Las implementaciones del estándar CORBA deben proveer un compilador que genere código en algún lenguaje específico a partir de los archivos de interfaces, dicho código es utilizado como punto de partida para el desarrollo de aplicaciones CORBA. Cada objeto CORBA es representado mediante una interfaz que define un conjunto de métodos e identificado por una referencia_de_objeto. Un cliente hace llamados a objetos remotos como si fueran objetos locales. El ORB es responsable de encontrar la implementación del objeto, comunicar la solicitud del cliente y en su caso, entregar la respuesta.

Java cuenta con un mecanismo (RMI) para la invocación de métodos remotos en aplicaciones desarrolladas enteramente en java. Cuenta también con su implementación del estándar CORBA, esto significa que cuenta con un compilador capaz de generar código java a partir de los archivos de interfaces IDL. El código generado se utiliza como base para la implementación del cliente y el servidor de la aplicación. Si se utiliza un compilador que genere código C++ a partir un archivo IDL y se implementa el servidor, este debe ser capaz de interoperar con un cliente implementado en java a partir del mismo archivo IDL, mediante el protocolo IIOP (Inter ORB Protocol) definido en el estándar.

Para especificar, diseñar y documentar aplicaciones desarrolladas con tecnología orientada a objetos, se utilizan lenguajes de modelado. La OMG propone un mantiene la especificación de lenguajes UML (Unified Modeling Language).

Lenguaje de Modelado Unificado - UML

UML es un lenguaje para especificar, construir, visualizar y documentar las piezas de un sistema de software orientado a objetos (OO). Mediante este lenguaje podemos crear modelos que nos permiten hacer abstracciones del problema y comprenderlo antes de empezar a implementarlo.

Un modelo incluye uno o más diagramas gráficos para representar los distintos puntos de vista de un sistema. Los diagramas utilizados son: diagramas de casos de uso, de clases, de secuencias, de colaboración, de actividades, de estados y de implementación.

Los diagramas de casos de uso sirven para especificar la funcionalidad y el comportamiento de un sistema mediante su interacción con los usuarios y/o otros sistemas.

Los diagramas de clases representan un conjunto de elementos del modelo que son estáticos tales como clases, paquetes, interfaces y las relaciones que se establecen entre ellos. Las clases describen un conjunto de objetos que comparte atributos, operaciones, métodos, relaciones y significado. Los paquetes se utilizan para organizar elementos en grupos. Las interfaces se utilizan para describir el comportamiento visible de cualquier elemento del modelo.

Los diagramas de secuencia muestran las interacciones entre un conjunto de objetos ordenadas según el tiempo en que tienen lugar. El objeto se puede crear o puede ser destruido durante la ejecución de la interacción. Un diagrama de secuencia es una forma de indicar el período durante el que un objeto está desarrollando una acción. Para representar las interacciones entre los objetos se utilizan mensajes. Existen distintos tipos de mensajes

según cómo se producen en el tiempo: simples, síncronos, y asíncronos. Los diagramas de secuencia permiten indicar cuál es el momento en el que se envía o se completa un mensaje.

Los diagramas de colaboración muestran la interacciones entre varios objetos y los enlaces que existen entre ellos. Representa las interacciones entre objetos organizadas alrededor de los objetos y sus vinculaciones. A diferencia de un diagrama de secuencias, un diagrama de colaboración muestra las relaciones entre los objetos, no la secuencia en el tiempo en que se producen los mensajes. Los diagramas de secuencias y los diagramas de colaboración contienen la misma información, pero la representan de forma diferente. La interacción entre los se representa mediante enlaces que indican que puede existir un intercambio de mensajes entre los objetos conectados.

Los diagramas de actividades se utilizan para mostrar el flujo de operaciones que se desencadenan en un procedimiento interno del sistema.

Los diagramas de estados representan la secuencia de estados por los que un objeto o un grupo de objetos pasan durante su tiempo de vida en respuesta eventos a recibidos. Cuando un objeto pasa de un estado a otro por la ocurrencia de un evento se dice que ha sufrido una transición, la transición puede ser interna si el estado del que parte el objeto es el mismo que al que llega y no se provoca un cambio de estado, en este caso se representan dentro del estado y no de la transición.

Los diagramas de implementación muestran los aspectos físicos del sistema. Incluyen la estructura del código fuente y la implementación. Los diagramas de componentes y los diagramas de plataformas o despliegue son tipos de diagramas de implementación. Un diagrama de componentes muestra la dependencia entre los distintos componentes de software y un diagrama de plataformas o despliegue muestra la configuración de los componentes hardware, los procesos, los elementos de procesamiento en tiempo de ejecución y los objetos que existen en tiempo de ejecución. En este tipo de diagramas intervienen nodos, asociaciones de comunicación, componentes dentro de los nodos y objetos que se encuentran a su vez dentro de los componentes. Un nodo es un objeto físico en tiempo de ejecución, es decir una máquina con memoria y capacidad de procesamiento, a su vez puede estar formada por otros componentes (*Ver el apéndice A para una referencia del lenguaje UML*).

I.V Problemática

Plantear un sistema que resuelva las necesidades de comunicación entre las aplicaciones de una empresa y que permita automatizar los procesos de negocio de la misma. Se requiere una forma de comunicación estándar que elimine la necesidad de interfaces punto a punto entre las aplicaciones y que permita la comunicación con aplicaciones de diversas tecnologías incluyendo aplicaciones heredadas. Se deben considerar las diferentes formas de comunicación síncronas y asíncronas para resolver las diferentes necesidades de comunicación. Es importante que se puedan incorporar nuevas aplicaciones al esquema de comunicación de forma sencilla, por lo anterior se requiere crear una infraestructura de comunicación entre aplicaciones.

Premisas

1. Los componentes a integrar pertenecen a diferentes aplicaciones.
2. Las aplicaciones fueron desarrolladas en diferentes lenguajes de programación.
3. La plataforma que da soporte a las aplicaciones (Hardware, Sistema Operativo y manejo de bases de datos) es distinta para cada una de las aplicaciones.
4. Se encuentra resuelta la comunicación entre las aplicaciones a nivel físico y de transporte (IP).

Características de la comunicación requerida

1. Comunicación entre aplicaciones de diversas plataformas y lenguajes.
2. Una aplicación que requiere un servicio, no tiene que saber los detalles de quien se lo está proporcionando.
3. Las aplicaciones no se preocupan por saber quien hace uso de los servicios que ofrecen.
4. Un servicio puede ser utilizado por más de una aplicación.
5. Los cambios en una aplicación, no impactan el esquema de comunicación con al resto.
6. No se pierde la información cuando alguna de las partes no esta disponible.
7. Debe existir la opción de comunicación síncrona y asíncrona.
8. Se pueden incorporar nuevos componentes de manera sencilla.

II ALTERNATIVAS DE SOLUCIÓN

A continuación se describen las diferentes formas de integración de sistemas que se han utilizado a lo largo del tiempo. Iniciamos con los mecanismos de Integración a nivel datos y posteriormente con la comunicación a nivel aplicativo. A final se hace un análisis y una propuesta de los mecanismos a utilizar para resolver el problema planteado.

II.I Integración a nivel datos

En la integración a nivel datos existe la posibilidad de que el cliente consulte la información en algún sitio remoto o bien, que la aplicación donde residen los datos envíe la información y posteriormente los datos sean manipulados para obtener un resultado (*figura 2.1*).

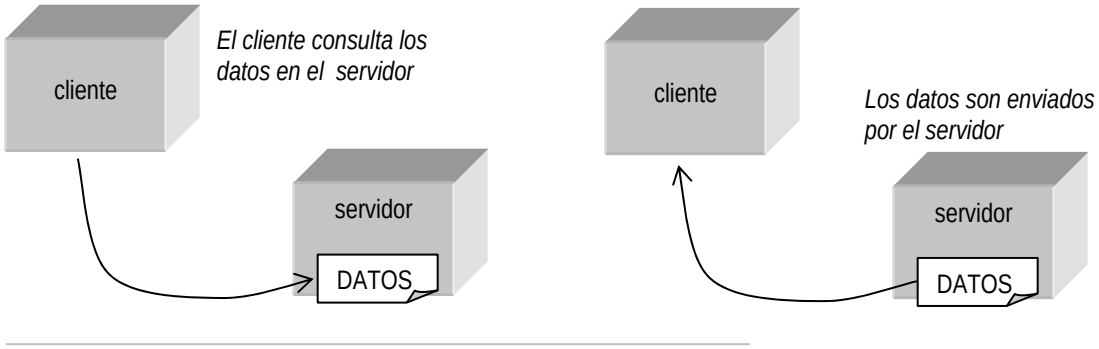


Figura 2.1. Integración a nivel datos.

Las diferentes formas de integración a nivel datos son: transferencia de archivos, acceso a archivos remotos, sentencias SQL y 'stored procedures'.

Transferencia de archivos

Cuando la información requerida se encuentra almacenada en algún archivo mantenido por una aplicación remota, existe la opción de transferirlo hasta la aplicación que la requiere. Esta es una forma sencilla de integración que resulta adecuada en casos muy específicos, por ejemplo cuando hablamos de archivos pequeños o con pocos cambios o bien cuando los datos son necesarios en momentos específicos, por ejemplo la información de las ventas del día requerida por la aplicación que realiza el corte.

Acceso a archivos remotos

Existen herramientas que permiten a una aplicación acceder a archivos remotos como si estos residieran en el mismo equipo, un ejemplo de esto es la utilización de NFS (Network File System). Un FS es una unidad organizacional donde se almacenan archivos, un NFS es un FS en un sistema remoto accesible a través de la red. Esta solución permite ocultar a la aplicación el hecho de que el archivo es remoto, de esta manera varias aplicaciones pueden acceder el archivo, ya sea para modificarlo o para obtener información.

La principal desventaja de acceder directamente a los archivos es que la información debe ser interpretada por la aplicación cliente. Generalmente se requiere que viaje gran cantidad de información para después ser procesada, este es un grave problema cuando los recursos de comunicación son limitados.

Sentencias SQL

Cuando las aplicaciones mantienen la información en una base de datos relacional, el acceso a la información se realiza a través de sentencias SQL. La información que viaja son las solicitudes de ejecución y el resultado. El uso de recursos de la red disminuye en comparación con la transferencia de todos los datos.

Stored Procedures

Típicamente se requiere la ejecución de varias sentencias para completar una operación de negocio. Si se agrupan estas sentencias en un procedimiento, se requerirá enviar un sólo mensaje para solicitar la ejecución del procedimiento en vez de solicitar la ejecución de cada una de las sentencias requeridas para completar la operación, con este esquema evitamos el envío de resultados parciales de la operación. El conjunto de sentencias SQL asociadas a una operación, se conoce como stored-procedure.

II.II Integración a nivel aplicativo

Cuando la funcionalidad de una aplicación se distribuye en diversos equipos interconectados entre si, estamos hablando de una aplicación distribuida. Las aplicaciones distribuidas toman ventajas de los diferentes recursos accesibles a través de la red como pueden ser procesamiento o almacenamiento. Una aplicación distribuida puede incorporar funcionalidades de aplicaciones existentes, esto tiene gran valor en una empresa donde se cuenta con aplicaciones legadas muy especializadas que sería muy costoso remplazar por aplicaciones de nueva generación.

Las aplicaciones distribuidas tienen que resolver el problema de la comunicación entre procesos corriendo en diferentes equipos, esto significa que los procesos se están ejecutando en diferentes espacios de direcciones. La forma de resolver la comunicación es con el uso de llamados a procedimientos remotos (RPC), con esto se pretende que los programadores puedan invocar procedimientos en equipos remotos de la misma forma que invocan procedimientos de forma local.

Procesos y comunicación entre procesos

Cuando un programa es leído del disco por el núcleo del SO y es cargado en memoria para ejecutarse, se convierte en un proceso. En el proceso se encuentra una copia del programa e información adicional para su manejo. Un proceso se compone de un segmento de texto que contiene el código compilado del programa, un segmento de datos que contiene los datos que se deben inicializar al arrancar el programa y un segmento de pila que contiene bloques para el manejo de llamados a funciones.

Cuando los procesos se encuentran en el mismo equipo, existen mecanismos de comunicación relativamente sencillos, tales como, semáforos, memoria compartida o colas

de mensajes. Cuando los procesos están en diferentes equipos se utilizan sockets o puertos para la comunicación. Para referirse a un equipo en la red, se utilizan direcciones que los identifican de manera única, a cada equipo con una dirección en la red se le conoce como nodo. Los mecanismos de comunicación entre procesos requieren contar con una referencia a una estructura de dirección, en lenguaje C la estructura se define de la siguiente forma:

```
<sys/socket.h>

struct sockaddr
{
     u_short   sa_family;
     char      sa_data[14];
}
```

sa_family indica la familia de protocolos a utilizar, la dirección está contenida en **sa_data** y el formato depende de la familia de protocolos usado.

Para la comunicación se utiliza el modelo cliente - servidor antes descrito. El flujo de comunicación entre el cliente y el servidor se ilustra en la figura 2.2.

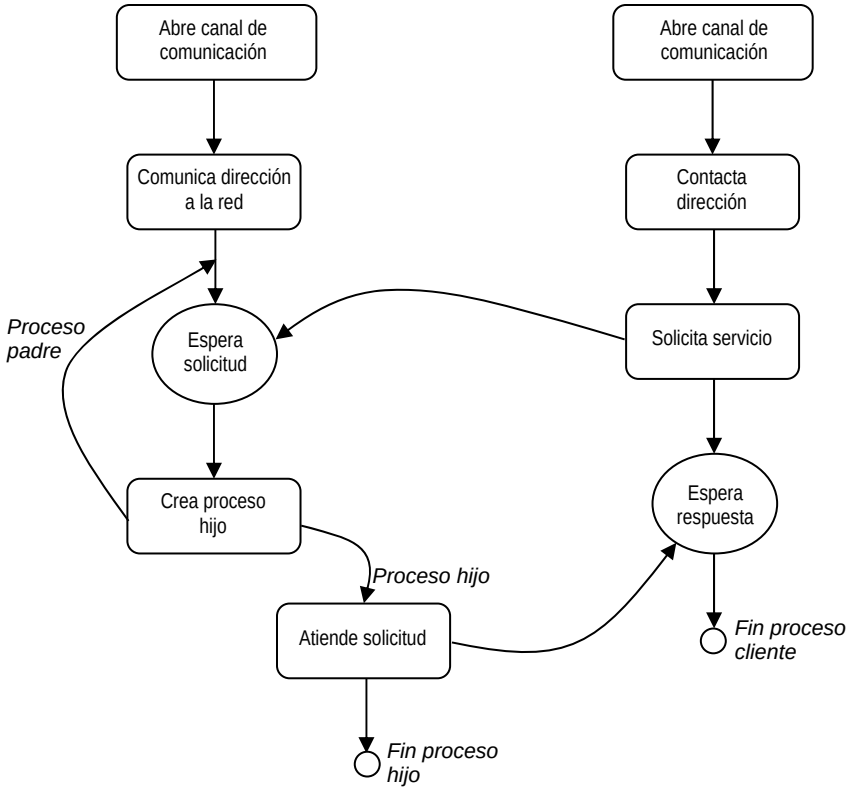


Figura 2.2. Comunicación por sockets, flujo de comunicación.

La secuencia de llamadas en una comunicación orientada a conexión, se muestra en la figura 2.3.

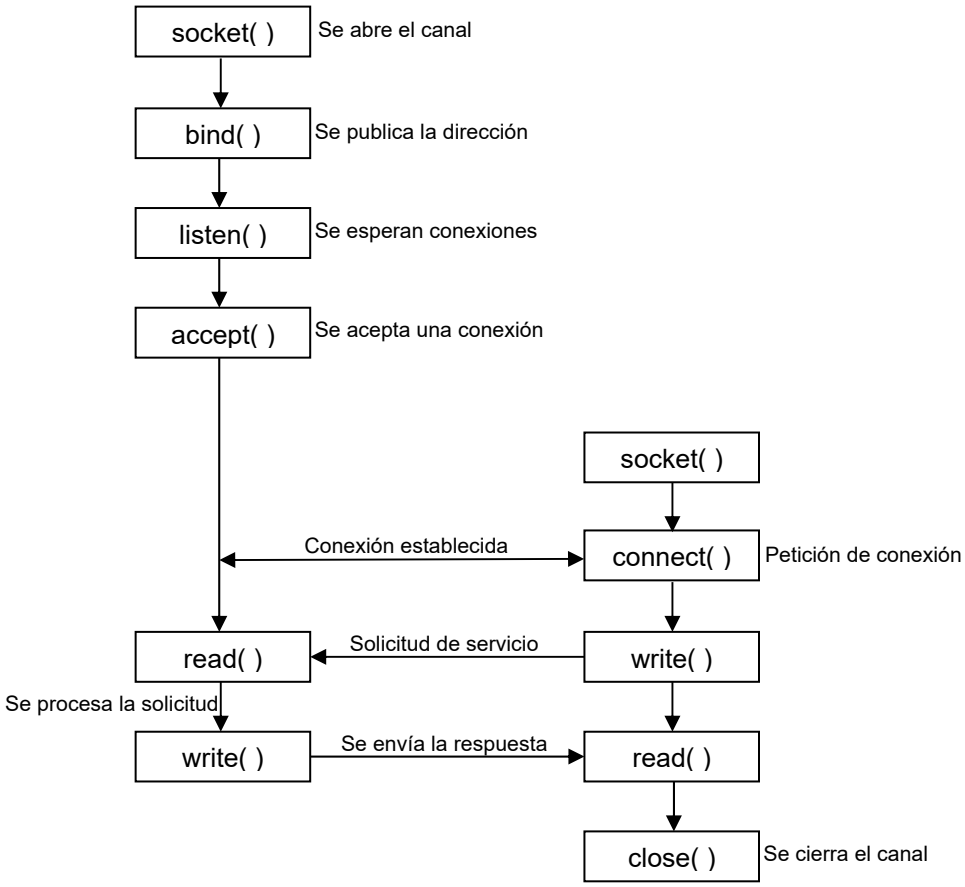


Figura 2.3. Comunicación por sockets, secuencia de llamadas.

A continuación se describen cada uno de los pasos del flujo de comunicación entre el cliente y el servidor.

Apertura del canal.
La llamada para abrir un canal bidireccional de comunicaciones se declara como sigue:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket (int af, int type, int protocol);
```

af(address family), especifica la familia a utilizar, las familias están definidas en **<sys/socket>,** las familias más comunes son las siguientes:

- **AF_UNIX** – Protocolos internos de UNIX, utilizada para conectar procesos ejecutándose en el mismo equipo.
- **AF_INET** – Protocolos de Internet, que utiliza los protocolos TCP o UDP.

- **AF_CCITT** – Norma X25 del CCITT.
- **AF_NS** – Protocolos NS de Xerox.

type, indica la semántica de comunicación y puede tener los siguientes valores:

- **SOCK_STREAM** – Orientado a conexión.
- **SOCK_DGRAM** – No orientado a conexión.

protocol especifica el protocolo que será utilizado. Por lo general las familias tienen asociados sólo un protocolo, si hubiera más de uno, este argumento es utilizado para especificarlo.

Si la llamada es correcta, socket devuelve un descriptor al conector.

Publicación de la dirección.
La llamada para ligar un conector con una dirección de red, se declara como sigue:

```
#include <sys/socket.h>
#include <sys/uf.h> /*Para la familia AF_UNIX. */
#include <sys/netinet.h> /*Para la familia AF_INET. */

int bind (int sfd, const void *addr, int addrlen);
```

bind hace que el conector cuyo descriptor es **sfd** se una a la dirección apuntada por **addr**. El formato de la dirección depende de la familia del conector. Para la familia **AF_UNIX**, se utiliza la estructura **struct sockaddr_un** definida en **<sys/uf.h>** y para la familia **AF_INET** la estructura **sockaddr_in** definido en **<sys/netinet.h>.** El valor de **addrlen** es el tamaño de la dirección y puede ser calculado con **sizeof**.

Las estructuras de las direcciones están definidas como sigue:

```
struct in_addr
{
     u_long         s_addr;        /* 32 bits. */
};
struct sockaddr_in
{
     short          sin_family;    /* AF_INET */
     u_shorts       sin_port;      /* 16 bits. */
     struct in_addr sin addr;      /* 32 bits. */
     char           sin_zero [8];  /* No usados. */
};
```

```
struct sockaddr_un
{
     short          sun_family;    /* AF_UNIX */
     char           sun_path [180];/* Ruta. */

};
```

Se esperan conexiones.
La llamada para indicar que se esperan conexiones se declara como sigue:

```
int listen (int sfd, int backlog);
```

Esta llamada habilita una cola asociada al conector descrito por **sfd**, en esta cola se alojan las peticiones de los clientes. La longitud de la cola se especifica en **backlog**. Esta llamada solo aplica a conectores **SOCK_STREAM** (orientados a conexión).

Petición de conexión.
La llamada para que el cliente inicie una conexión con el servidor se declara como sigue:

```
#include <sys/socket.h>
#include <sys/uf.h> /*Para la familia AF_UNIX. */
#include <sys/netinet.h> /*Para la familia AF_INET. */

int connect (int sfd, const void *addr, int addrlen);
```

Normalmente la llamada permanece bloqueada hasta que se completa la conexión. Si la conexión no se puede realizar inmediatamente y el conector tiene activo el modo **0_DELAY** (se activa mediante una llamada **fcntl**), se devuelve el control inmediatamente y se indica que se ha producido un error.

Aceptación de conexión.
La llamada utilizada por el servidor para aceptar peticiones se define como sigue:

```
#include <sys/socket.h>

int accept (int sfd, const void *addr, int addrlen);
```

Esta llamada es para conectores orientados a conexión. **accept** extrae la primera petición de conexión que hay en la cola creada por **listen**, posteriormente crea un nuevo conector con las mismas propiedades que **sfd** y reserva el descriptor **nsfd** para él. Si no hay peticiones de conexión y el conector esta en modo bloqueante (**0_NDELAY**), la llamada permanece bloqueada hasta recibir una nueva petición. En caso no-bloqueante, se devuelve el control y se indica que se ha producido un error.

Lectura de un conector.
Existen varias llamadas que permiten leer datos o mensajes de un conector establecido. la llamada **readv** puede utilizarse para leer datos de un fichero o de un conector. La declaración es la siguiente:

```
#include <sys/uio.h>

ssize_t readv (int fildes,
               const struct iovec iov,
               size_t iovcnt);
```

readv lee los datos del conector **fildes** y coloca la información en **iov[0], iov[1],..., iov[iovcnt1].** Cada elemento del arreglo tiene una estructura **iovec** que se define de la siguiente manera:

```
struct iovec{
     caddr_t iov_base;   /*Dirección inicial*/
     int iov_len         /*Tamaño en bites*/
```

Si se llega al final del archivo o se completan los bloques de memoria, termina la ejecución y regresa el número de bytes leídos.

Las llamadas **recv, recvfrom, y recvmsg** sólo pueden leer datos de un conector, la declaración de las funciones es la siguiente:

```
#include <sys/socket.h>

int recv (int sfd, void *buf, int len, int flags);
int recvfrom (int sfd, void *buf, int len, int flags,
                void *from, int fromlen);
int recvmsg(intsfd, struct msghdr msg[], int flags);
```

sfd representa el conector del cual se van a leer los datos, **buf** es un apuntador de la memoria donde se van a escribir los datos y **len** es el número máximo de bytes en **buf**. En la llamada **recvfrom**, la dirección del conector origen de los datos se escribe en la estructura apuntada por **from**, a la hora de llamar la función **fromlen** contiene el tamaño de la estructura apuntada por **from**, a la salida contendrá el tamaño real de la dirección leída.

El argumento msg es un arreglo de cabeceras de mensajes con la siguiente estructura:

```
struct msghdr{
     caddr_t        msg_name;
     int            msg_namelen;
     struct iovec   msg_iov;  /*Arreglo para los datos leídos*/
     int            msg_iovlen;    /*Tamaño del arreglo*/
     caddr_t        msg_accrights; /*Derechos de acceso*/
     int            msg_accrightslen;
}
```

Esta estructura se emplea tanto para leer datos de un conector, como para escribirlos en él, **msg_name** y **msg_len** se utilizan para conocer la dirección origen o destino de un conector sin conexión, **msg_iov** es el arreglo de bloques de memoria en que se escribirán los datos

leídos o de donde se recogerán los datos a enviar, **msg_accrights** es de donde se recogerán los derechos de acceso enviados con el mensaje.

El argumento **flags** en las llamadas, puede tener los bits **MSG_PEEK** y **MSG_OOB**. El bit de **MSG_PEEK** indica que el dato leído del conector puede obtenerse en la siguiente lectura, se utiliza para revisar si hay datos en el conector, **MSG_OOB** se utiliza para leer datos a los que se les dará un carácter urgente en el envío o recepción.

Escritura a un conector.
Para la escritura a un conector se tienen funciones homologas a las que se utilizan para leer. La definición es la siguiente:

```
#include <sys/uio.h>

     ssize_t writev (int fildes,
                     const struct iovec *iov,
                     size_t iovcnt);

#include <sys/socket.h>

     int send (int sfd, void *buf, int len, int flags);
     int sendto (int sfd, void *buf, int len, int flags,
               void *to, int tolen);
     int sendmsg (int sfd, struct msghdr msg [ ], int flags);
```

Cierre de un canal.
Para cerrar un conector se utiliza la llamada **close**, esta llamada cierra el conector en sus dos sentidos.

Como podemos observar, la integración mediante socket es a nivel de comunicaciones. Se puede utilizar como una base para empezar a crear lógica de integración. De manera natural no es una herramienta que maneje la tecnología orientada a objetos, aún cuando lenguajes como java poseen su propia implementación. El manejo de socket es particularmente útil para el intercambio de información punto a punto entre sistemas (*Ver el apéndice B, en la parte 1. Sockets, para ejemplos de programas de comunicación).*

El Modelo RPC

El modelo RPC describe la manera en que se comunican dos procesos, que están siendo ejecutados en diferentes nodos de la red, para coordinar actividades. Este modelo se basa en el modelo de llamados a procedimientos en los lenguajes de programación, la diferencia es que la llamada se realiza entre un proceso cliente y un proceso servidor ejecutándose en nodos distintos de la red.

Cuando un programa invoca una función o procedimiento local (*figura 2.4*), la dirección de retorno se almacena en una pila y se transfiere el control a la función. La ejecución de la función sólo significa un cambio en la dirección del hilo de ejecución, el retorno de la función

saca el registro de la pila y entrega la dirección al contador del programa, de esta manera se continua con la ejecución en la línea que sigue a la llamada.

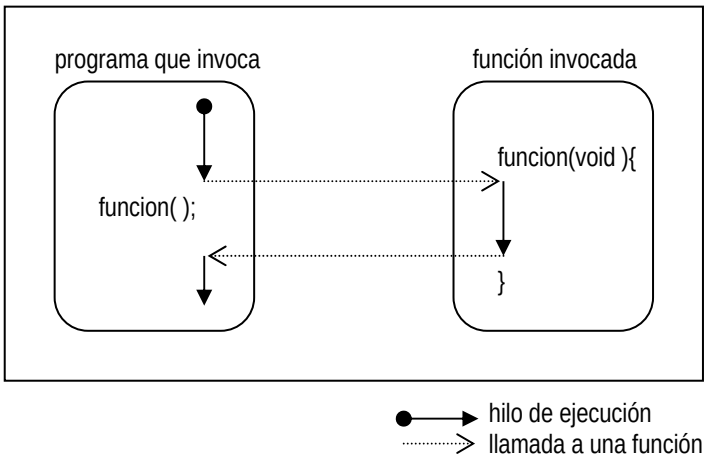


Figura 2.4. Llamada a una función local.

Para invocar una función situada en el espacio de direcciones de un proceso que reside en otra máquina, es necesario crear un nuevo hilo de ejecución en la máquina remota. En este caso el hilo de ejecución local permanece bloqueado hasta que el hilo de ejecución remota concluye. (*figura 2.5*)

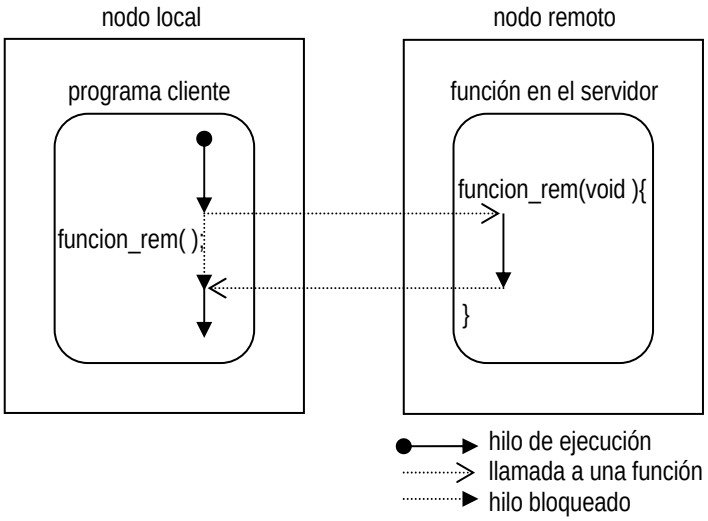


Figura 2.5. Llamada a una función remota.

El programa cliente debe compilarse con una pieza de código adicional que se encarga del manejo de la llamada remota a la que se llama stub-cliente. El cliente realiza una llamada local y espera una respuesta. Del lado del servidor, las funciones que pueden ser invocadas de manera remota, deben ser compiladas con código adicional llamado stub-servidor.

Cuando llega una invocación por la red, el stub-servidor es el que recibe la llamada y hace una invocación local al servidor.

La operación básica de una llamada a un procedimiento remoto (*figura 2.6*) inicia cuando la aplicación cliente realiza una llamada a un stub-cliente. El stub-cliente recibe y regresa argumentos al procedimiento que hace la llamada, los argumentos son instancias de parámetros ya sea de entrada o de salida. Posteriormente se convierten los argumentos de entrada de una representación local, a una representación común, se crea un mensaje con estos argumentos y se hace una llamada al runtime del cliente. El runtime, usualmente es un conjunto de librerías con rutinas de soporte para el stub-cliente.

El runtime transmite el mensaje con los argumentos de entrada a un runtime en el servidor. El runtime en el servidor usualmente es un conjunto de librerías con rutinas de soporte para el stub-servidor. El runtime hace una llamada al stub-servidor, éste toma los parámetros de entrada y los convierte de una representación común, a una representación local para así hacer una llamada a la aplicación servidora.

La aplicación servidora procesa la llamada y regresa el resultado al stub-servidor. Se realiza el mismo procedimiento de conversión de argumentos para hacer llegar la respuesta a la aplicación cliente.

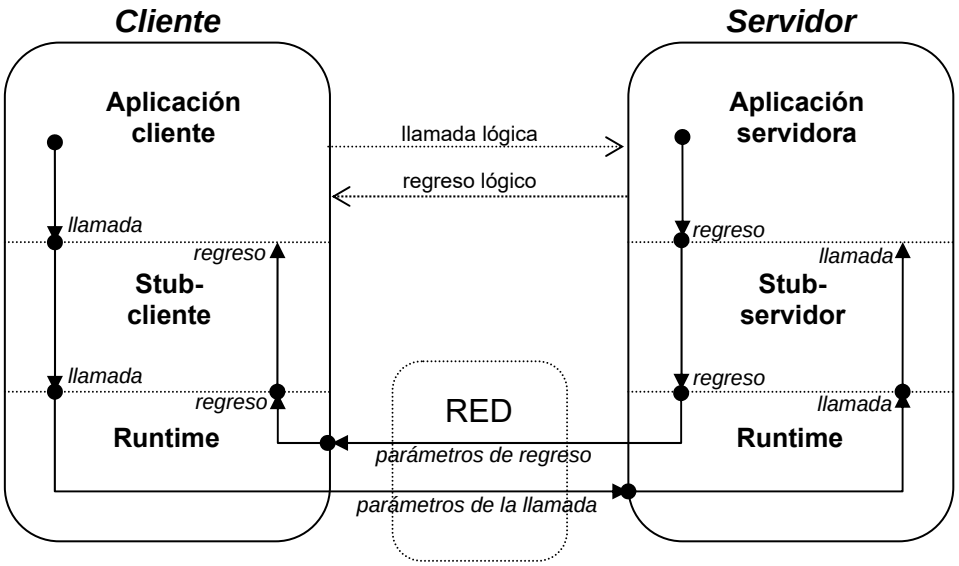


Figura 2.6. Modelo RPC.

El modelo RPC incluye el concepto de hilos de ejecución, de esta manera un cliente puede iniciar una invocación y continuar con otro procesamiento en otro hilo de ejecución. La aplicación debe reconocer que se ha concluido con la ejecución de un RPC por alguna técnica de poleo o alguna interrupción de software.

Existen herramientas (ej. TI-RPC(ONC) de Sun: rcpgen) que permiten generar archivos para el cliente y el servidor. Dichos archivos deben ser modificados por el programador y se debe insertar el código de la implementación de las funciones. Los archivos son generados a partir de un archivo de especificación que contiene los prototipos de las funciones (*Ver el apéndice B, en la parte 2. RPC, para más detalle de la conversión de una llamada local a una llamada remota).*

II.III Software de conectividad y de servicios distribuidos

El software de conectividad y servicios distribuidos permite la interacción de procesos corriendo en diferentes máquinas en la red. Este software se encuentra entre la aplicación y el sistema operativo y los servicios de red (*figura 2.7*).

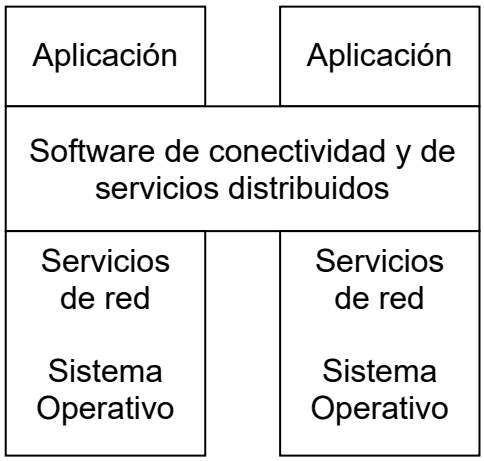


Figura 2.7. Localización del software de conectividad y de servicios distribuidos.

Este tipo de software, también conocido como *Middleware,* ofrece a las aplicaciones las siguientes características:

- Localización transparente a través de la red
- Independencia de los servicios de red
- Confiabilidad
- Escalabilidad sin perdida de función

Se han realizado diferentes trabajos que ofrecen alguna o todas las características antes mencionadas, entre ellos podemos encontrar los siguientes:

- **DCE**: *Distributed Computing Environment* (Ambiente Distribuido de Cómputo) de *Open Software Foundation*
- **COM/DCOM***: Component Object Model/Distribiuted COM* de Microsoft
- **Java RMI**: *Remote Method Invocation* con Java de Sun Microsystems

- **CORBA**: *Common Object Request Broker Arquitecture* (Arquitectura común de ORB's) de la OMG (*Object Management Group)*

DCE

DCE: Distributed Computing Environment (Ambiente Distribuido de Cómputo) es un conjunto de servicios de sistemas que proporcionan un ambiente distribuido con el objetivo principal de resolver problemas de interoperabilidad en ambientes de red heterogéneos. Fue desarrollado y es mantenido por la *Open System Foundation*.

Cuenta con un código fuente o implementación de referencia en la cual están basados todos los productos DCE. La implementación de referencia es independiente de la red y de los sistemas operativos y proporciona una arquitectura en la cual se pueden incluir nuevas tecnologías. La idea de DCE es que la implementación de referencia proporcione tecnología que pueda ser utilizada en parte como servicios individuales o como una infraestructura integral completa. Soporta la construcción e integración de aplicaciones cliente - servidor intentando esconder al usuario la complejidad inherente al procesamiento distribuido. La arquitectura de DCE se muestra en la figura 2.8.

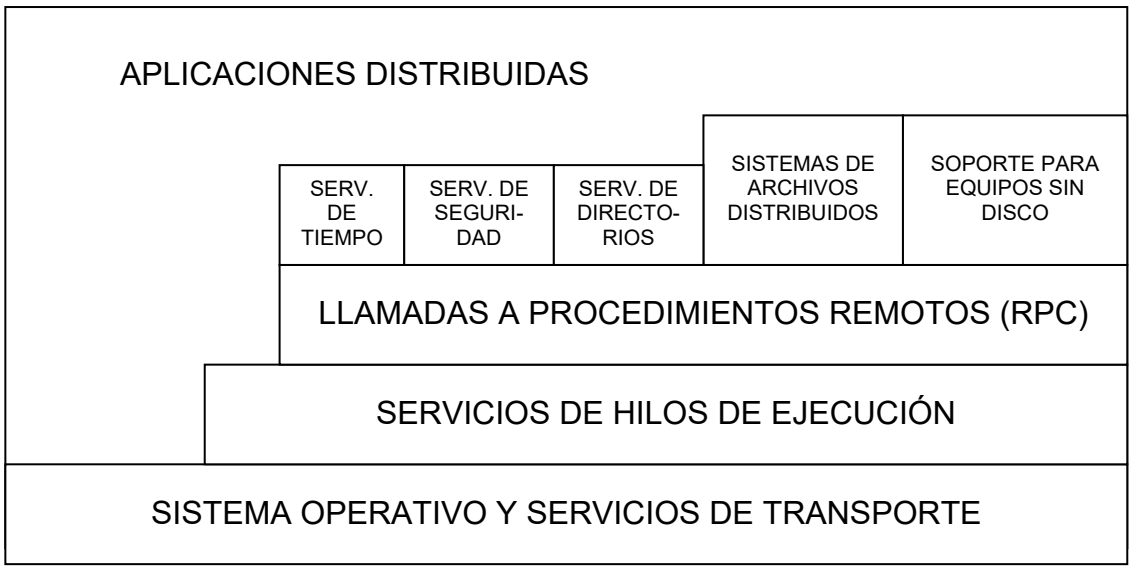


Figura 2.8. Arquitectura de DCE.

Los servicios que DCE ofrece se dividen en dos categorías:

- Servicios distribuidos fundamentales: proporcionan herramientas que permiten a los desarrolladores de software crear servicios de cómputo distribuidos. Los servicios básicos incluyen: Servicios para el manejo de hilos de ejecución, llamados a procedimientos remotos, servicios de tiempo, servicios de seguridad (autenticación, autorización y manejo de cuentas de usuarios) y servicios de directorios (identificación de recursos distribuidos).

- Servicios de datos compartidos: proporcionan capacidades construidas sobre los servicios distribuidos fundamentales, estos servicios no requieren programación adicional y facilitan el uso de la información. En esta categoría se encuentran: Sistemas de archivos distribuidos y soporte a equipos sin disco (diskless support).

DCE soporta estándares OSI: *Open System Interconnect* tales como CCITT X.500, ROSE: *Remote Operations Services Element* y ACSE: *Association Control Service Element*. DCE también soporta estándares de Internet tales como TCP/IP, DNS: *Domain Name System* y NTP: *Network Time Protocol*.

DCE está escrito en C estándar y utiliza interfaces de sistemas operativos estándares como POSIX y X/OPEN, esto lo hace portable a diferentes plataformas. Internamente trabaja con el modelo cliente - servidor adecuado para aplicaciones trabajando con este modelo.

La mayoría de los servicios DCE están optimizados mediante la estructuración de los sistemas de cómputo en células (un conjunto de nodos) que se consideran como una unidad. La comunicación en una misma célula es optimizada y relativamente segura y transparente. La comunicación entre células requiere procesamiento más especializado y complejo y requiere más esfuerzo de programación.

Cuando se utiliza el servicio de hilos de ejecución (thread), el programador debe estar consiente de la sincronización y de los datos compartidos. Debe considerarse la complejidad del manejo de hilos de ejecución si se va utilizar este servicio.

La versión 1.0 del código fuente de DCE fue liberado en 1992, en 1993 se encontraban disponibles kit´s de desarrollo poco robustos principalmente para plataformas UNIX. En 1996 se liberó la versión 1.2.1 con soporte para lenguaje de definición de interfaces (IDL) para C++.

DCE no fue escrito para ser completamente orientado a objetos, las interfaces estándares utilizadas por DCE están definidas en C. Para las aplicaciones orientadas a objetos es más complejo y costoso utilizar servicios no orientados a objetos. En 1996 se empezó a trabajar en extensiones orientadas a objetos para DCE.

COM/DCOM

Es una especificación e implementación desarrollada por Microsoft, provee un marco de referencia para la integración de componentes. Soporta los conceptos de Interoperabilidad y reusabilidad de objetos distribuidos, permite a los desarrolladores construir sistemas mediante la unión de componentes de terceros que se comuniquen vía COM: *Component Object Model*.
COM define una API: *Application Programming Interface* (Interfaz de programación de la aplicación) que permite la creación de componentes para usarse en la integración de aplicaciones personalizadas o para permitir a los componentes interactuar. Los componentes deben adherirse a una estructura binaria especificada por Microsoft. En la medida que los componentes se adhieran a esta estructura binaria, los componentes escritos en diferentes lenguajes, pueden ínteroperar.

DCOM: *Distributed COM*, es una extensión de COM que permite la interacción de componentes en la red, mientras que los procesos COM pueden correr en diferentes espacios de memorias, pero en la misma máquina, con DCOM los componentes operando en diferentes plataformas en la red pueden interactuar, siempre y cuando DCOM este disponible dentro del ambiente.
En conjunto COM y DCOM están incluidos en un runtime que permite tanto accesos locales como remotos.
COM y DCOM se consideran tecnologías de bajo nivel que permite la interacción de componentes, hay servicios de aplicación de más alto nivel construidos sobre esta tecnología, por ejemplo: OLE, ActiveX y MTS.

OLE proporciona servicios de *linking* y *embedding* utilizados en la creación de documentos generados por múltiples fuentes. ActiveX permite a los componentes ser incluidos en sitios Web. MTS proporciona servicios empresariales como transacciones y seguridad.

COM+ es una evolución de COM que integra servicios MTS y encolamiento de mensajes y hace la programación más sencilla mediante la integración con lenguajes de Microsoft tales como Visual Basic, Visual C++ y J++, escondiendo parte de la complejidad de la programación COM.

COM es una especificación de compatibilidad binaria y una implementación asociada que permite a los clientes invocar servicios proporcionados por componentes. Los servicios implementados por objetos COM son expuestos mediante un conjunto de interfaces que representan el único punto de contacto entre los clientes y el objeto (*figura 2.9*).

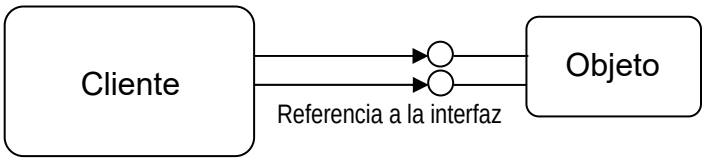


Figura 2.9. Comunicación entre un cliente y un servidor mediante una interfaz.

COM define una estructura binaria para las interfaces entre el cliente y el objeto que proporciona las bases para la interoperabilidad entre componentes. Los objetos COM pueden ser escritos en cualquier lenguaje que soporte la estructura binaria COM de Microsoft. Un objeto COM puede soportar cualquier número de interfaces (*figura 2.10*), cada interfaz proporciona un grupo de métodos relacionados.

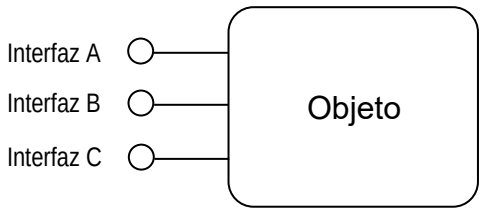


Figura 2.10. Múltiples interfaces de un objeto.

Los objetos y las interfaces COM se especifican mediante un lenguaje de definición de interfaces (IDL) de Microsoft, que es una extensión del IDL estándar de DCE. Cada objeto y cada interfaz deben tener un identificador único.

Una vez que se define una interfaz, ésta se considera inmutable, no se pueden agregar métodos ni se deben cambiar los existentes. Siguiendo esta regla se evitan problemas de compatibilidad de versiones, de esta manera no se afecta a los clientes dependientes de estas interfaces. Si se requiere agregar funcionalidad a un componente, se debe definir una nueva interfaz que soporte la funcionalidad existente más las mejoras.

Los objetos COM se ejecutan en un servidor, un servidor puede soportar múltiples objetos COM. Existen tres formas en que un cliente puede acceder a los objetos COM en un servidor (*figura 2.11*):

1. El cliente puede ligar librerías en el servidor. El cliente y el servidor se ejecutan en el mismo proceso. La comunicación se lleva a cabo mediante llamados a funciones.
2. El cliente puede acceder a un servidor corriendo en diferente proceso, pero en el mismo equipo. La comunicación se resuelve mediante mecanismos de comunicación entre procesos.
3. El cliente puede acceder a un servidor remoto. La comunicación entre el cliente y el servidor se resuelve mediante RPC´s. El mecanismo de acceso a servidores remotos se conoce como DCOM.

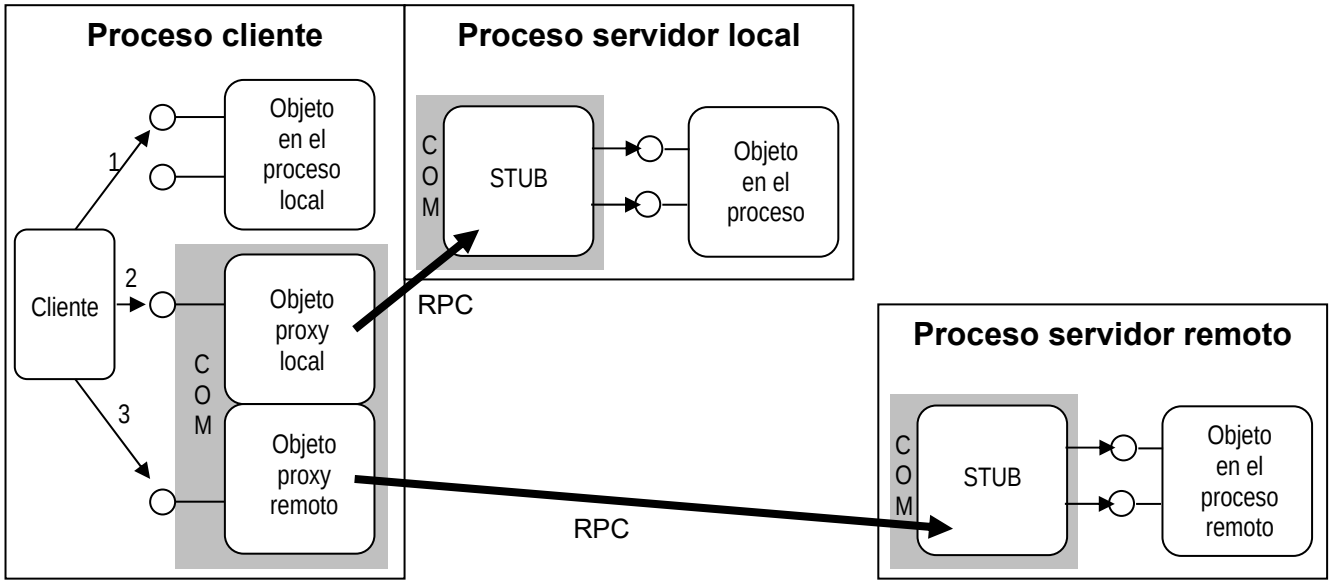


Figura 2.11. Arquitectura de COM - DCOM.

Si el cliente y el servidor están en el mismo proceso, compartir datos es relativamente simple. Sin embargo, cuando el proceso servidor está separado del proceso cliente, ya sea en el mismo servidor o en un servidor remoto. COM debe formatear y empacar los datos que serán compartidos, este proceso se lleva a cabo mediante un objeto-proxy que se comunica con un objeto-stub diseñado para una interfaz en particular. El flujo de comunicación entre el proceso cliente y el proceso servidor se muestra en la figura 2.12.

COM crea los stubs en el proceso del servidor y los proxys en los procesos de los clientes. El proxy proporciona el apuntador a la interfaz. El cliente llama las interfaces del servidor mediante el proxy, el cual formatea y empaca los parámetros y los pasa al stub en el servidor. El stub desempaca los parámetros y hace la llamada al objeto servidor. Cuando se completa la llamada, el stub formatea y empaca los valores de regreso y los pasa al proxy, éste a su vez los regresa al cliente. El mismo proceso se utiliza si el cliente y el servidor están en la misma máquina o en distinta, sin embargo es distinto la implementación interna del proceso de paso de parámetros.

Dado un archivo IDL, el compilador de IDL de Microsoft crea un código base para el proxy y el stub que realizan las tareas de formateo y empacamiento de parámetros.

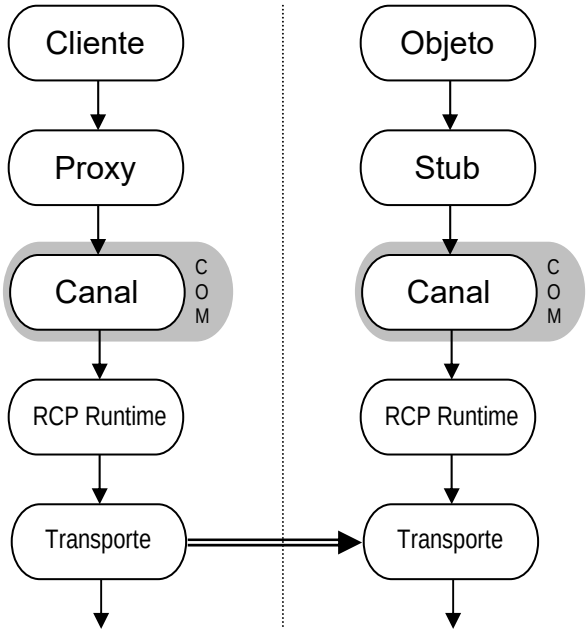


Figura 2.12. Flujo de comunicación entre un cliente y un servidor.

Cuando un cliente desea utilizar un objeto, lleva a cabo los siguientes pasos (*figura 2.13*):

1. Invoca un API para crear una instancia de un objeto COM.
2. COM localiza la implementación del objeto e inicializa un proceso servidor para el objeto.
3. El proceso servidor crea un objeto y regresa una referencia a la interfaz del objeto.
4. El cliente utiliza la referencia para interactuar con el objeto.

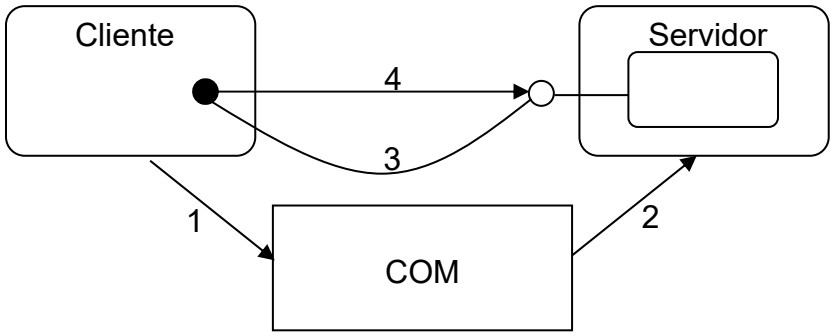


Figura 2.13. Comunicación entre un cliente y un servidor.

Los objetos en COM no tienen identidades, un cliente pregunta por un objeto COM de algún tipo y no por un objeto en particular. Cada vez que un objeto COM es solicitado, se crea una nueva instancia.

COM incluye interfaces y funciones API que exponen servicios del sistema operativo así como otros mecanismos necesarios para un ambiente distribuido tales como servicios de nombres y eventos.

Los servicios que incluye la tecnología COM son los siguientes:

- Almacenamiento estructurado y persistencia: los objetos COM requieren de un mecanismo para almacenar datos cuando no están corriendo, el proceso de almacenar datos para un objeto, se conoce como hacer persistente al objeto. COM soporta la persistencia de objetos mediante el almacenamiento estructurado.
- Monikers: los clientes requieren en ocasiones conectarse exactamente a la misma instancia de un objeto y con el mismo estado que en la ocasión anterior, para ello se utilizan los monikers.
- Objetos conectables: En ocasiones los objetos requieren un mecanismo para notificar a los clientes de la ocurrencia de un evento. COM permite a los objetos definir interfaces de notificación que son implementadas por los clientes. Esta interfaz permite una comunicación bi-direccional entre el cliente y el objeto.

Una de las desventajas de COM es que es difícil de utilizar, se requiere un conocimiento profundo de la tecnología. Además no es suficientemente robusto para desarrollos empresariales ya que no integra servicios como seguridad, transacciones, confiabilidad en la comunicación y balanceo de cargas. Esto se trata de resolver en COM+ y MTS.

Otro aspecto a considerar es la plataforma, COM - DCOM esta diseñado para trabajar en ambientes Windows. Algunas compañías han liberado DCOM para OS/390, HP-UX, SUN Solares y Linux, sin embargo es muy difícil conseguir soporte. No se cuenta con herramientas para procesamiento en tiempo real, ni de alta confiabilidad.

Debido a que COM - DCOM está basado en un formato binario nativo, los componentes no son independientes de la plataforma, por lo que deben ser recompilados para una plataforma específica o se debe contar con un intérprete. Por supuesto que este tiene la ventaja de mejor desempeño y aprovechamiento de las capacidades nativas de la plataforma.
Existen herramientas no muy costosas que permiten la creación y el acceso a componentes COM para Windows, podemos mencionar Visual Basic y Visual C++. Considerar herramientas para otras plataformas es mucho más costoso y complicado ya que en estás plataformas no se elimina la complejidad inherente a las aplicaciones distribuidas. No se puede pensar que con el uso de la tecnología COM/DCOM se reduce la experiencia requerida para el desarrollo de aplicaciones distribuidas o el uso de múltiples hilos de ejecución, a pesar del la fuerte organización con la que cuenta Microsoft para ofrecer soporte.

RMI

El mecanismo de Invocaciones Remotas de Java permite hacer un llamado a un método en un objeto que existe en otro espacio de direcciones ya sea en el mismo equipo o en un equipo remoto, este mecanismo es parecido a un llamado RPC pero orientado a objetos. Existen tres procesos que participan en una llamada RMI: el cliente, el servidor y el objeto de

registro. El cliente hace una invocación a un método en un objeto remoto en el servidor. El objeto registro es un servicio que asocia nombres a los objetos, una vez que un objeto es registrado, se puede utilizar su nombre para accederlo.

Java define un modelo de objetos distribuidos en el que los objetos remotos son aquellos que implementan interfaces remotas y cuentan con métodos que pueden ser invocados por otras maquinas virtuales en cualquier equipo accesible en la red. Una invocación a un método en un objeto remoto, tiene la misma sintaxis que una invocación en un objeto local.

Un sistema RMI tiene una arquitectura de tres capas independientes entre si; la capa stub - skeleton, la capa de referencias remotas y la capa de transporte (*figura 2.14*).

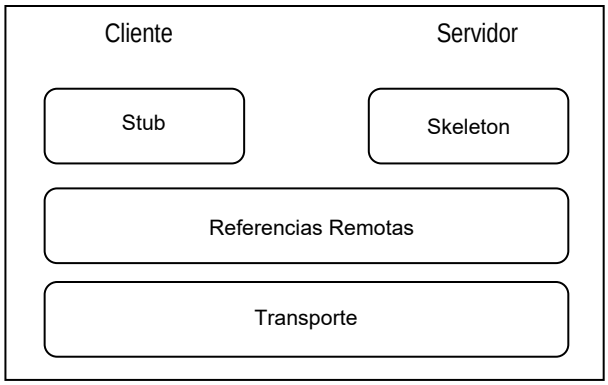


Figura 2.14. Arquitectura de un sistema RMI.

Las tres capas RMI están posicionadas arriba de una Máquina Virtual Java, así una aplicación RMI cuenta con las características de Java: Recolección de basura, mecanismos seguros de carga de clases, etc.; las capas de la aplicación están por encima de RMI.

Para escribir una aplicación RMI el primer paso es la definición de la interfaz que describa los métodos que serán invocados por los clientes. La interfaz debe extender **`java.rmi.Remote`** para indicar que es un servicio remoto.

```
import java.rmi.*;
public interface MyService extends java.rmi.Remote
{
// Definición de los métodos
     public valor_de_retorno nombre_del_metodo1 ( parametros)
          throws RemoteException;

     public valor_de_retorno nombre_del_metodo2 ( parametros)
          throws RemoteException;
     // ...
}
```

En la implementación de las interfaces se debe definir un constructor por omisión que se utiliza para manejar las excepciones **`java.rmi.RemoteException`**

```
public MyServiceServer () throws RemoteException
{
     super();
}
```

La implementación debe contener un método **`main()`** que será responsable de crear la instancia del servicio y registrarlo.

```
public stratic void main (String args[]) throws Exception
{
     //Creación de la instancia del servicio
     MyServiceServer srv = new MyServiceServer();

     //Registro del servicio
     Naming.bind("MyService",srv);

     System.out.println("Servicio registrado...");
}
```

El cliente llama al objeto registro para obtener la referencia del objeto remoto e invocar los métodos remotos, toda la parte de comunicaciones queda oculta al programador. La dirección URL contiene el hostname donde se encuentra el servicio y el nombre. Como resultado de la siguiente llamada **`Naming.lookup`** se obtiene un instancia del servicio sobre la cual se pueden hacer invocaciones.

```
     MyService service = (MyService)Naming.lookup
          ("rmi://" + args[0] + "/MyService");

     System.out.println ("Resultado de la llamada:" +
          service.metodo(parametros));
```

El mecanismo RMI es adecuado cuando se trabaja con aplicaciones desarrolladas puramente en java. Se pueden integrar aplicaciones corriendo en diferentes plataformas de forma sencilla (*Ver el apéndice B, en la parte 3. RMI, para más detalle de una invocación a un método remoto java).*

CORBA

La Arquitectura CORBA (Common Object Request Broker Architecture) propuesta por la **OMG** permite el desarrollo de aplicaciones de objetos distribuidos, incorpora objetos desarrollados en diversos lenguajes de programación, corriendo en diversas plataformas.

El ORB (Object Request Broker) es un ente responsable de manejar las peticiones de servicios entre clientes e implementaciones de objetos (*figura 2.15*).

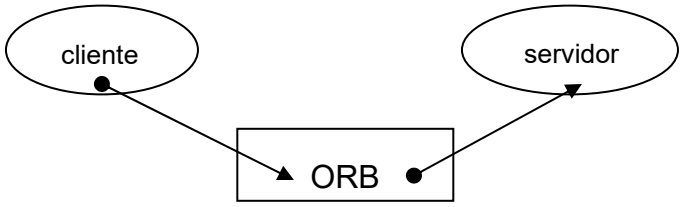


Figura 2.15. Comunicación entre un cliente y un servidor mediante un ORB.

El cliente es una entidad capaz de solicitar un servicio. La Implementación de un objeto es responsable de que se lleven a cabo los servicios solicitados, incluye ejecutar ciertos cálculos para obtener un resultado, cambiar el estado del sistema o solicitar nuevos servicios.

El ORB recibe la solicitud del cliente y la envía a la implementación del objeto, cuando se completa la petición, el control y los parámetros de salida son regresados al cliente. La implementación del ORB puede estar residente en el cliente, en la implementación del objeto o puede localizarse en un servidor dedicado. La información necesaria para especificar un objeto dentro de un ORB es conocida como *Referencia de Objeto*.

Una operación es una entidad identificable que denota un servicio indivisible que puede ser proporcionado. El hecho de solicitar el servicio de la operación se denomina *Invocar la Operación*. Las operaciones pueden o no incluir parámetros y pueden ser invocados de forma síncrona o síncrona diferida. Una excepción es una indicación de que la operación no se llevó a cabo con éxito.

Mediante una interfaz se describen las posibles operaciones que un cliente puede solicitar a la Implementación del objeto. Las interfaces pueden ser específicas o dinámicas para la petición en cuestión (*figura 2.16*). Las interfaces específicas que utiliza el cliente para comunicarse con el ORB se denominan stubs y son independientes de la localización de la implementación del objeto. La implementación del objeto recibe la petición a través de una interfaz específica conocida como skeleton o de una interfaz dinámica.

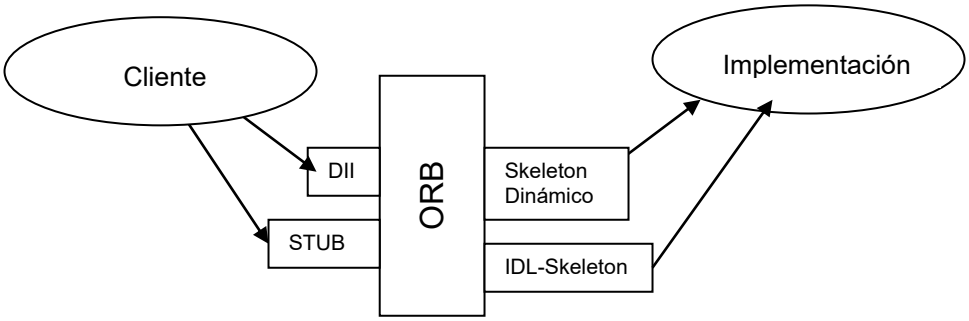


Figura 2.16. Interfaces estática y dinámicas.

Para la definición de interfaces se utiliza un lenguaje IDL: *Interface Definition Language* (Lenguaje de Definición de Interfaces). Se definen interfaces independientes de la plataforma y del lenguaje. La figura 2.17 muestra como se utiliza un compilador IDL para generar los archivos stub y skeleton que son utilizados por el cliente y el servidor.

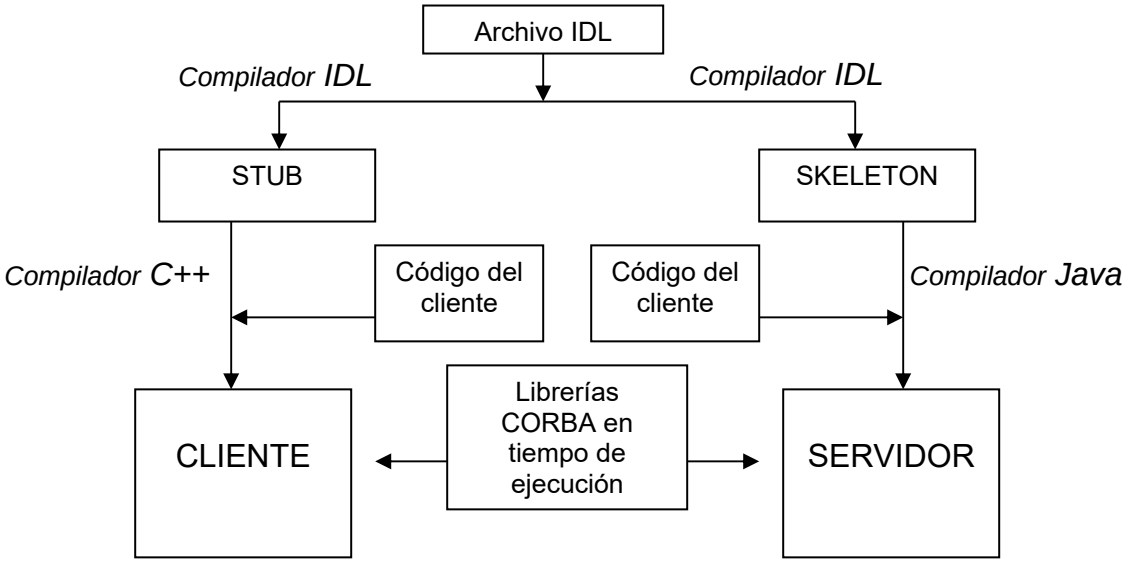


Figura 2.17. Uso de un Compilador IDL.

La secuencia de eventos que ocurren desde que el cliente solicita un servicio y el último evento derivado de esa solicitud se conoce como *Petición de Servicio*. Los métodos contienen el código ejecutado para proporcionar un servicio. Una 'Máquina de Ejecución' se encarga de interpretar los métodos y provocar que se lleve a cabo el código descrito. La ejecución del método se llama *Activación del Método*. La implementación del objeto incluye la definición de los métodos.

La implementación del objeto accede a los servicios que proporciona el ORB mediante un adaptador que depende del tipo de servicios que requiere. Para la localización y activación de las implementaciones de los objetos, el ORB utiliza un repositorio de implementación. La información de las interfaces disponibles en tiempo de ejecución de almacenan en un repositorio de interfaces. (*Ver el apéndice B, en la parte 4. CORBA, para ejemplos de archivos generados por un compilador IDL.*

II.IV Análisis de las alternativas de integración

Las diferentes alternativas de comunicación entre sistemas pueden ser adecuadas para diferentes necesidades de integración.

La comunicación a nivel datos no permite agregar ninguna lógica a la integración por lo que debemos restringir su uso a casos específicos. Cuando utilizamos la comunicación directa entre aplicaciones, no tenemos la suficiente flexibilidad ya que la lógica para establecer la

comunicación reside en las aplicaciones mismas. En la tabla 2.18 se comparan las características de las principales formas de comunicación entre aplicaciones.

Aspecto	COM/DCOM	Java RMI	CORBA
Lenguaje de desarrollo	C++, VB, Java	Java	Cualquiera
Plataformas	Windows	No Mac	Varias
Estándar	Casi-abierto	Cerrado	Abierto
Orientado a objetos	No	Casi	Si
Definición de Interfaces remotas	MS IDL, ODL	Java	IDL
Facilidad de desarrollo	Difícil	Muy fácil	Fácil

Tabla 2.18. Comparación entre tecnologías para ambientes distribuidos.

Cuando se tiene un ambiente heterogéneo, la alternativa es utilizar alguna implementación del estándar CORBA, sin embargo la comunicación entre dos aplicaciones basada en CORBA es esencialmente síncrona. Para resolver la problemática planteada, no es suficiente tomar alguna de las formas de comunicación y empezar a crear interfaces, es necesario crear un sistema que sirva de mediador entre las aplicaciones, que permita desacoplarla y que ofrezca diferentes opciones de comunicación entre ellas.

III EL SERVICIO DE EVENTOS

Una solicitud CORBA es esencialmente síncrona, la solicitud va a un objeto específico y tanto el cliente como el servidor deben estar disponibles para que la petición sea exitosa.

Un Servicio de Eventos permite el intercambio asíncrono de mensajes entre objetos CORBA. Los objetos CORBA en un ambiente de eventos son conocidos ya sea como proveedores, que producen eventos, o como consumidores, que reciben eventos. Los datos del evento son pasados a través de parámetros definidos como mejor convenga.

III. I Descripción de un Servicio de Eventos

Un Servidor de eventos es un módulo de software que se encarga de recibir eventos y entregarlos a los clientes interesados. Los clientes deben informar al servidor que están interesados en algún tipo de evento, deben suscribirse al evento. Es trabajo del servidor registrar y mantener la información necesaria para poder notificar a los suscriptores cuando un evento de interés ocurra.

Con la ayuda de un servidor de eventos se logra una comunicación desacoplada entre las aplicaciones, las aplicaciones no tienen que preocuparse por saber quien recibe los eventos generados y de igual forma los clientes no saben quien provee los servicios que utiliza.

III. II Arquitectura de comunicación en un esquema de eventos

La comunicación en un esquema de eventos puede ser iniciada por el cliente o por el proveedor de un servicio.

Una aplicación cliente inicia la comunicación cuando requiere un servicio que es proporcionado por una aplicación remota. El cliente envía un mensaje de solicitud de un servicio al Servicio de Eventos, éste debe poder identificar a la aplicación servidora, solicitar el servicio y entregar la respuesta a la aplicación cliente (*figura 3.1*).

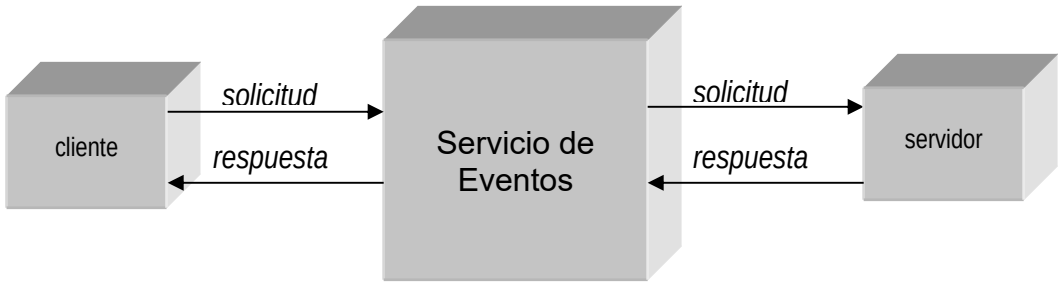


Figura 3.1. Cliente – servidor.

Una aplicación proveedora inicia la comunicación cuando ocurre un evento que debe ser notificado a una o más aplicaciones consumidoras. La aplicación publicadora envía un mensaje a las aplicaciones consumidoras para notificarles de la ocurrencia de un evento (*figura 3.2*).

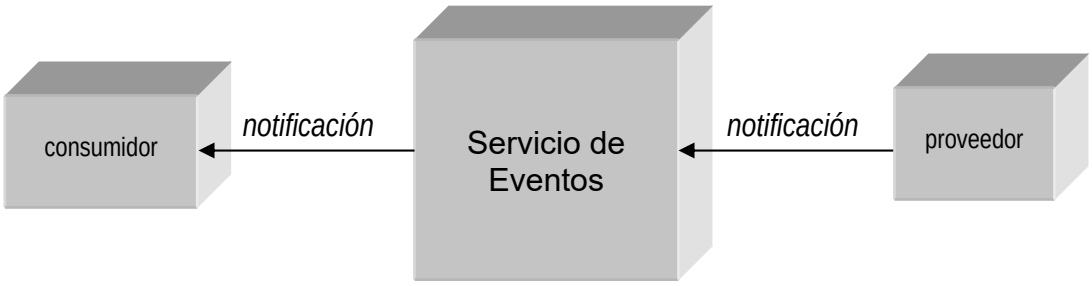


Figura 3.2. Proveedor – consumidor.

III. III Estándares actuales para un Servicio de Eventos

El estándar CORBA de la OMG incluye los siguientes servicios:

- Servicio de Eventos
- Servicio de Notificación
- Servicio de Nombres
- Servicio de Localización
- Servicio de Bitácora
- Servicio de Tiempo

Los que se refieren al manejo de eventos son el Servicio de Eventos y el Servicio de Notificación que se describen a continuación.

Servicio de Eventos de la OMG

En el modelo normal de comunicación CORBA entre aplicaciones distribuidas, una aplicación cliente invoca una operación IDL en un objeto específico en un servidor. El cliente espera la ejecución en el servidor y recibe la respuesta. Existe solamente un cliente y un servidor y ambos deben estar disponibles para que la invocación sea exitosa.

El servicio de eventos, contenido en la especificación CORBA de la OMG, define un modelo de comunicación que permite a una aplicación enviar un mensaje que será recibido por varios objetos. Un evento se genera en un proveedor de eventos y es transferido a uno o más consumidores de eventos. Los consumidores y proveedores de eventos están completamente desacoplados, es decir los proveedores no saben quien recibió el evento y los consumidores no saben quien los generó.

En este modelo se utilizan canales de eventos que tienen las siguientes funciones:

- Permitir a los consumidores registrarse a ciertos eventos.
- Aceptar eventos de los proveedores.
- Reenviar los eventos a los consumidores suscritos.

Tanto los consumidores como los proveedores se conectan a los canales de comunicaciones y no directamente (*figura 3.3*). Desde el punto de vista de un proveedor, un canal de eventos es un sólo consumidor. Desde el punto de vista del consumidor, el canal de eventos es un sólo proveedor.

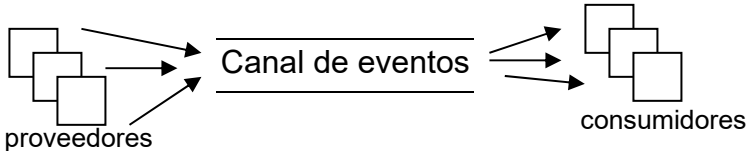


Figura 3.3. Canal de eventos.

Cualquier número de proveedores pueden enviar eventos a cualquier número de consumidores a través de un canal de eventos, no existe ninguna relación entre el número de proveedores y consumidores. Nuevos proveedores y consumidores pueden ser agregados fácilmente a un canal, y un proveedor o un consumidor se puede conectar a más de un canal de eventos.

Los proveedores, consumidores y canales de eventos, implementan interfaces IDL claramente definidas que soportan los pasos requeridos para transferir eventos en un sistema distribuido.

Un evento ocurre cuando un proveedor invoca una operación claramente definida en un objeto en el canal de eventos de la aplicación. El canal de eventos propaga el evento mediante una invocación a una operación similar en un objeto en cada uno de los servidores-consumidores.

CORBA especifica dos formas de iniciar la transferencia de eventos entre proveedores y consumidores. El modelo push y el modelo pull. En el modelo push, los proveedores inician la transferencia de eventos; en el modelo pull, los consumidores inician la transferencia mediante solicitudes de eventos.

En el modelo push, un proveedor genera eventos y los envía activamente a los consumidores (*figura 3.4*).

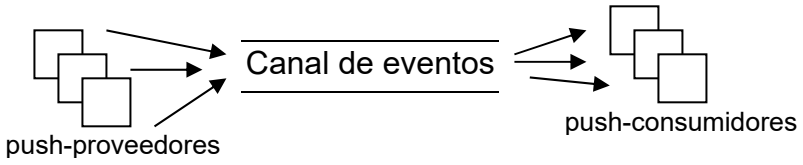


Figura 3.4. Modelo push.

El proveedor inicia la transferencia de un evento mediante la invocación de una operación IDL en un objeto en el canal de comunicación, el canal de eventos invoca una operación similar en cada consumidor registrado en el canal de eventos.

En el modelo pull, un consumidor activamente solicita que un proveedor genere un evento. Cuando la solicitud del consumidor llega al proveedor, este genera y regresa la información al consumidor (*figura 3.5*).

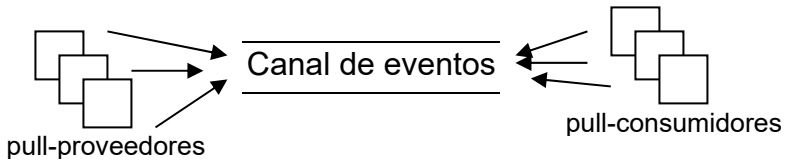


Figura 3.5. Modelo pull.

Un consumidor inicia la transferencia de un evento invocando una operación IDL en un objeto en el canal de eventos. El canal de eventos invoca una operación similar en un objeto proveedor. El resultado es regresado por el proveedor al canal de eventos y a su vez al consumidor que inicio la transferencia.

Los modelos se pueden mezclar de manera que podemos tener push-proveedores y pull-consumidores iniciando eventos en un mismo canal.

El servicio de eventos de CORBA mapea un evento a una secuencia de llamadas a operaciones completada exitosamente. Las operaciones y la secuencia de llamadas están claramente definidas para los dos modelos, los datos relacionados con los eventos son parámetros de las operaciones o valores de regreso y son específicos de cada aplicación.

La especificación del servicio de eventos de CORBA define que la comunicación de eventos puede ser no_tipificada tipificada.

En una comunicación no_tipificada, un evento es propagado por una serie de llamadas a invocaciones genéricas **`push()`** o **`pull()`**. La operación **`push()`** tiene un parámetro con los datos del evento, este parámetro es de tipo *any*, que permite pasar cualquier tipo de dato definido en IDL entre proveedores y consumidores. La operación **`pull()`** no tiene parámetros, pero transmite datos del evento en su valor de regreso, que es también de tipo *any*. Las aplicaciones proveedoras y consumidoras deben estar de acuerdo en el contenido de los parámetros y valor de regreso.

En una comunicación tipificada, un programador define interfaces IDL para aplicaciones especificas a través de las cuales los eventos son propagados. No se utilizan las operaciones **`push()` y `pull()`**. La interfaz definida se utiliza por proveedores y consumidores para la comunicación de eventos. Las operaciones definidas en las interfaces pueden contener parámetros definidos de cualquiera de los tipos de datos IDL. *Refiérase al apéndice C para una referencia del Lenguaje de Definición de Interfaces.*

El servicio de eventos define un conjunto de interfaces IDL que soportan los modelos **push** y **pull** para comunicación no_tipificada y tipificada. Mediante estas interfaces se define el papel que desempeñan los proveedores, consumidores y canales de eventos de cada modelo. Las operaciones en dichas interfaces permiten la propagación de eventos a los proveedores y consumidores.

Se definen también un conjunto de interfaces de administración que permiten a los proveedores y consumidores registrarse con un canal de eventos para permitir la transferencia de eventos entre ellos.

Un proveedor se conecta a un canal de eventos para indicar que desea transferir eventos a consumidores a través de dicho canal. Un consumidor se conecta a un canal de eventos para indicar que está interesado en algunos de los eventos proporcionados a través de ese canal. Cuando un proveedor o consumidor ya no desea enviar o recibir eventos, se puede desconectar del canal de eventos. En algunos casos el canal de eventos puede desconectar a algún proveedor o consumidor explícitamente.

A continuación se describen las interfaces del modelo push y del modelo pull contenidas en el estándar del servicios de eventos de la OMG.

Modelo push

Se definen cuatro interfaces que soportan la conexión y desconexión de canales de eventos utilizando el modelo push.

PushSuppier
PushConsumer
ProxyPushConsumer
ProxyPushSupplier

Las interfaces **PushSupplier** y **ProxypushConsumer** permiten a los proveedores proporcionar eventos al canal de eventos. Las interfaces **PushConsumer** y **ProxyPushSupplier** permiten a los consumidores recibir eventos de los canales de eventos.

```
Module CosEventComm {
   Exception Disconnected {
   };

   interface PushConsumer {
      void push (in any data) raises (Disconnected);
      void disconnect_push_consumer ();
   };

   interface PushSupplier {
      void disconnect_push_supplier ();
   };
};
```

```
module CosEventChannelAdmin {
   exception AlreadyConnected {
   };

   exception Typeerror {
   };

   interface ProxyPushConsumer:CosEventComm::PushConsumer {
      void connect_push_supplier (
         in CosEventComm::PushSupplier push_supplier)
         raise (AlreadyConnected);
   };

   interface ProxyPushsupplier:CosEventComm::PushSupplier {
      void connect_push_consumer (
      in CoseventComm::PushConsumer push_consumer)
      raises (AlreadyConnected, TypeError);
   };
   …
};
```

Para conectarse a un canal de eventos, el proveedor obtiene una referencia a un objeto del tipo *ProxyPushConsumer* en el canal. El proveedor invoca la operación **connect_push_supplier()** y le pasa una referencia de un objeto *PushSupplier* como parámetro de la operación. Si el *ProxyPushConsumer* ya está conectado al *PushSyupplier,* **connect_push_supplier()** alcanzará la excepción **AlreadyConnected**.

Para conectarse a un canal de eventos, el consumidor obtiene una referencia a un objeto del tipo *ProxyPushSupplier* en el canal. El proveedor invoca la operación **connect_push_consumer()** y le pasa una referencia de un objeto *PushConsumer* como parámetro de la operación. Si el *ProxyPushSupplier* ya está conectado al *PushConsumer*, **connect_push_consumer()** alcanzará la excepción **AlreadyConnected** (*figura 3.6*).

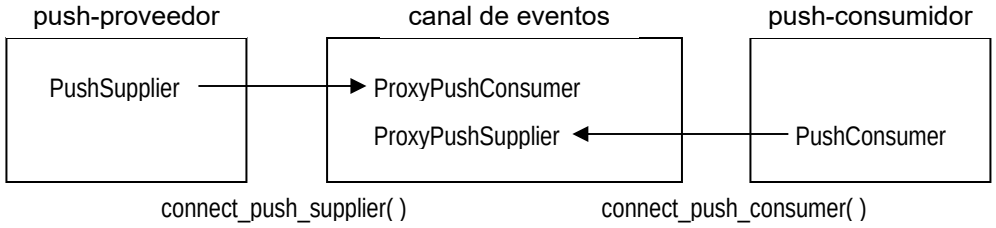


Figura 3.6. Conexión de un proveedor y un consumidor en el modelo push.

Una vez que el proveedor y el consumidor se encuentran conectados, se puede iniciar la transferencia de eventos mediante el canal de eventos (*figura 3.7*).

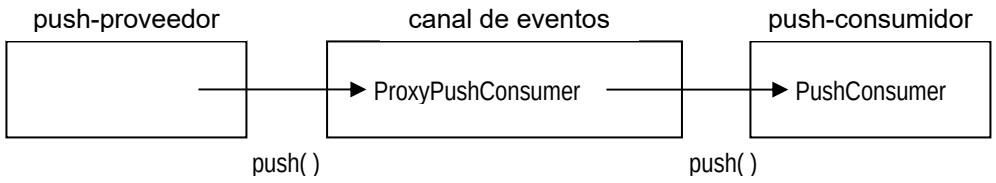


Figura 3.7. Transferencia de eventos en el modelo push.

Modelo pull

Al igual que para el modelo push, se definen interfaces similares para el modelo pull.

PullSupplier
PullConsumer
ProxyPullConsumer
ProxyPullSupplier

Las interfaces **PullConsumer** y **ProxyPullSupplier** permiten a los consumidores solicitar eventos a un canal de eventos. La interfaces **PullSupplier** y **ProxyPullConsumer** permiten a un canal de eventos, solicitar eventos a los proveedores.

```
Module CosEventComm {
   Exception Disconnected {
   };

   interface PullSupplier {
      any pull() raises (Disconected);
      any try_pull(out boolean has_event)
         raises(Disconnected);
      void disconnect_pull_supplier ();
   };

   interface PullConsumer {
      void disconnect_pull_consumer ();
   };
};

module CosEventChannelAdmin {
   exception AlreadyConnected {
   };

   exception Typeerror {
   };

   interface ProxyPullSupplier:CosEventComm::PullSupplier {
      void connect_pull_consumer (
         in CosEventComm::PushConsumer pull_consumer)
         raise (AlreadyConnected);
```

```
   };

   interface ProxyPullConsumer:CosEventComm::PullConsumer {
      void connect_pull_supplier (
      in CoseventComm::PullSupplier push_supplier)
      raises (AlreadyConnected, TypeError);
   };
   …
};
```

Para conectarse a un canal de eventos, el consumidor obtiene una referencia a un objeto del tipo *ProxyPullSupplier* en el canal. El proveedor invoca la operación **`connect_pull_consumer()`** y le pasa una referencia de un objeto *PullConsumer* como parámetro de la operación. Si el *ProxyPullSupplier* ya está conectado al *PullConsumer*, **`connect_pull_consumer()`** alcanzará la excepción **`AlreadyConnected`**.

Para conectarse a un canal de eventos, el proveedor obtiene una referencia a un objeto del tipo *ProxyPullConsumer* en el canal. El proveedor invoca la operación **`connect_pull_supplier()`** y le pasa una referencia de un objeto *PullSupplier* como parámetro de la operación. Si el *ProxyPullConsumer* ya está conectado al *PullSyupplier*, **`connect_pull_supplier()`** alcanzará la excepción **`AlreadyConnected`** (*figura 3.8*).

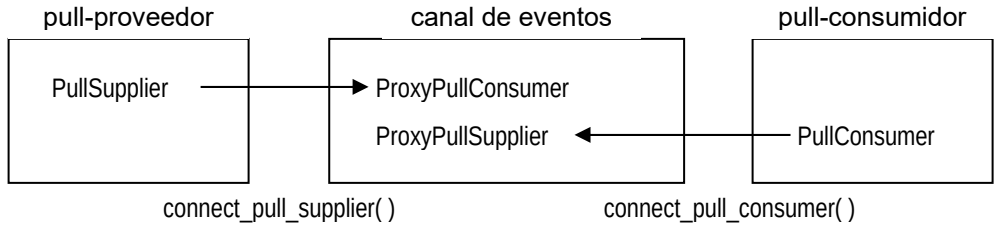


Figura 3.8. Conexión de un consumidor y un proveedor en el modelo pull.

Una vez que el proveedor y el consumidor se encuentran conectados, se puede iniciar la transferencia de eventos mediante el canal de eventos (*figura 3.9*).

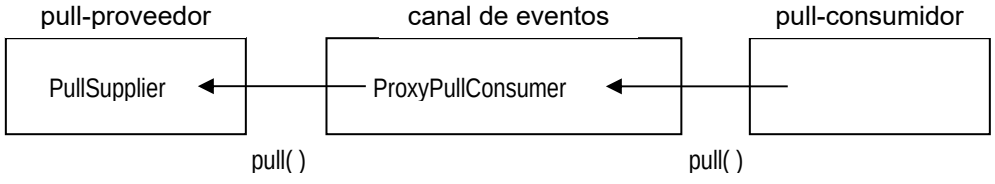


Figura 3.9. Transferencia de eventos en el modelo pull.

Servicio de Notificación de la OMG

Al igual que el servicio de eventos, el servicio de notificación permite la comunicación entre clientes y servidores de manera desacoplada mediante el uso de canales de eventos.

Tanto los proveedores como los consumidores se conectan al canal de comunicación y no directamente (*figura 3.10*).

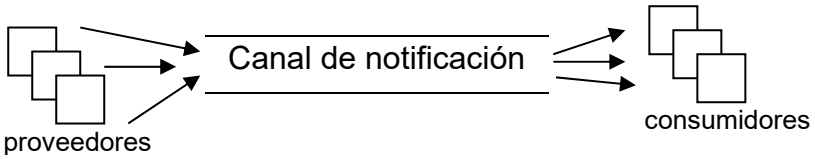


Figura 3.10. Canal de notificación.

Cualquier número de proveedores y consumidores se pueden conectar a un canal de comunicación. Un proveedor o un consumidor se puede conectar a más de un canal de comunicación.

El servicio de notificación extiende el concepto de mensajería del servicio de eventos con las siguientes características:

- Calidad de servicio que permite definir prioridades y tiempo de vida en un canal de eventos.
- Filtrado de eventos y suscripción que permite a los consumidores recibir sólo los eventos de su interés e informar a los proveedores la demanda de eventos.
- Publicación de eventos que permite a los proveedores informar a un canal de eventos los eventos que puede proporcionar.

El proveedor, el consumidor y el canal de notificación pueden ser implementados como clientes o como servidores. Típicamente el proveedor se implementa como un cliente y el canal de eventos y los consumidores se implementan como servidores.

Servicio de Nombres

Un servicio de nombres se utiliza para mapear referencias de objetos a nombres. Permite a los clientes localizar objetos utilizando nombres que son independientes del objeto al que hacen referencia. Esto permite hacer cambios en la implementación de un objeto o en su localización de manera transparente a los clientes.

El servicio de nombres provee un repositorio para las referencias de objeto, las aplicaciones lo utilizan para obtener referencias iniciales a objetos. Los nombres se organizan en una gráfica con una estructura jerárquica (*figura 3.11*). El nodo padre es el contexto inicial del nombre, este nodo al igual que los otros tiene una o más referencias asociadas, las referencias pueden ser a objetos o a otros contextos de nombres.

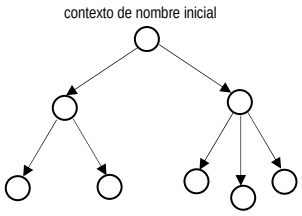


Figura 3.11. Gráfica de estructura de nombres.

El servicio de nombres utiliza operaciones y tipos definidos en tres interfaces: *NamingContext*, *NamingContextExt* y *BindingIterator*.

Una secuencia de nombres especifica la ruta de un contexto de nombre a otro o a una referencia a un objeto, cada nombre representa un nodo en la ruta. Un nombre se compone de dos cadenas: **id,** es el identificado del componente y **kind,** que es opcional y se utiliza para diferenciaciones posteriores. Ambos miembros se utilizan en la resolución de nombres.

```
module CosNaming {
     typedef string Istring;
     struct NameComponent {
          Istring id;
          Istring kind;
     }
     typedef secuence<NameComponent> Name;
     …
};
```

Para obtener la referencia a un objeto se siguen los siguientes pasos:

1. Se obtiene la referencia al contexto inicial de nombres.
2. Se construye una estructura con el nombre asociado al objeto a resolver
3. Se obtiene la referencia al objeto.

Para obtener la referencia inicial al contexto de nombres se utiliza un código similar al siguiente:

```
global_orb = org.omg.CORBA.ORB.init(args, null);
org.omg.CORBA.Object obj =
     global_var.resolve_initial_references(“NameService”);
...
org.omg.CosNaming.NamingContextExt root_cxt;
     rootcxt = org.omg.CosNaming.NamingContextExtHelper.narrow(obj);
```

Para construir la estructura del nombre, se seleccionan los miembros **id** y **kind** de cada elemento del nombre.

```
org.omg.CosNaming.NameComponent [ ] name = new NameComponenet[2];
```

```
name [0] = new NameComponent (“Nombre1”, “ “);
name [1] = new NameComponent (“Nombre2”, “ “);
```

Finalmente, para resolver el nombre se utiliza una instrucción **`resolve()`** en la siguiente forma:

```
org.omg.CORBA.Object obj;
...
obj =root_cxt.resolve(name);
```

Para construir una jerarquía de nombres se siguen los siguientes pasos:

1. Se obtiene la referencia al contexto inicial de nombres.
2. Se construye la primer capa de contextos de nombres.
3. Se asocia cada nombre a la referencia inicial.
4. Se agregan nuevos contextos de nombres subordinados a la primer capa.

Las siguientes líneas de código crean un nuevo contexto de nombres y lo asocian a la contexto inicial.

```
org.omg.CosNaming.NamingContext nuevo_cxt =
     root_cxt.new_context();

org.omg.CosNaming.Naming.NameComponent [ ] name =
     new NameComponent[1];
name [0] = new NameComponent(“nuevo”, “ “);

root_cxt.bind_context(name, nuevo_cxt);
```

Para el mantenimiento del Servicio de Nombres se deben remover las asociaciones de los contextos que ya no se utilicen y posteriormente destruirlos.

Un gráfica de nombres se puede distribuir en múltiples Servicios de Nombres en diferentes equipos. Con un Servicio de Nombres federado se obtienen los siguientes beneficios:

- Se minimiza el impacto en la falla de un servidor.
- Se distribuye el procesamiento y se pueden balancear las cargas.
- Se puede manejar almacenamiento persistente en diversos equipos permitiendo la escalabilidad.
- Administración descentralizada.

IV PROPUESTA DE UN SERVICIO DE EVENTOS

En el presente capítulo se hace una propuesta para un Servicio de Eventos, se inicia con los requerimientos funcionales y técnicos y posteriormente se plantea el diseño del componente de software. Para los requerimientos funcionales y el diseño se utilizan diagramas UML (Lenguaje Unificado de Modelado). Se presentan también las alternativas que se tienen en cuanto a lenguajes de programación se refiere.

La figura 4.1 muestra un diagrama donde se ve el sistema desde un punto de vista general, podemos ver que el Servicio de Eventos será responsable de la comunicación entre un grupo de aplicaciones.

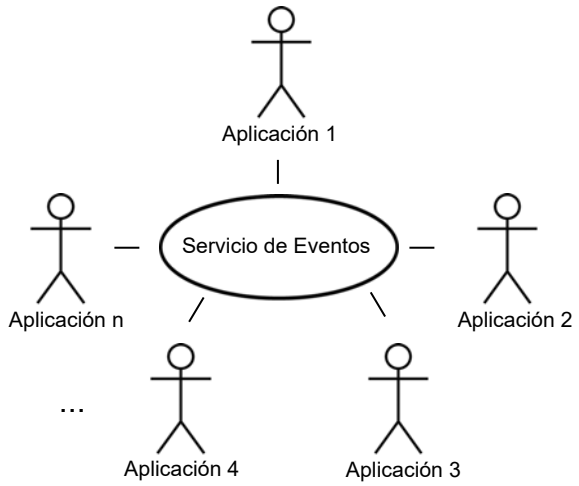


Figura 4.1. Diagrama general del Servicio de Eventos.

IV. I Requerimientos funcionales

Actores

Los principales actores que interactúan con el Servicio de Eventos son las aplicaciones que desean integrarse al esquema de comunicación. Las aplicaciones pueden jugar uno de los cuatro roles en la figura 4.2.

Adicionalmente participa el Operador del Servicio de Eventos que se encarga de la administración del sistema en general. El resumen de los actores del sistema lo encontramos en la tabla 4.3.

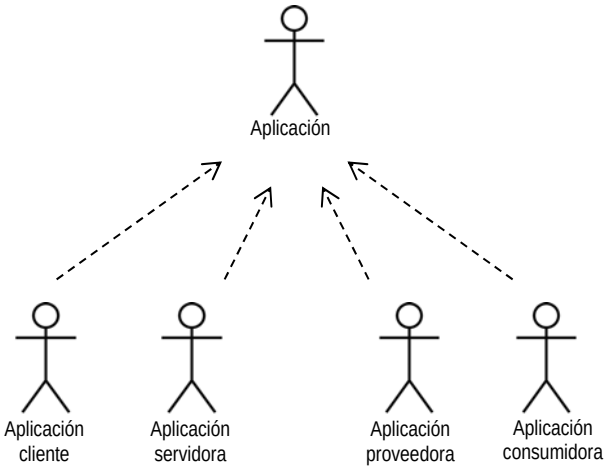


Figura 4.2. Rol que pueden desempeñar las aplicaciones.

Actor	Descripción
Aplicación	Cualquier aplicación que interactúa con el Servicio de Eventos
Aplicación cliente	Aplicación que solicita servicios al Servicio de Eventos.
Aplicación servidora	Aplicación que proporciona servicios a las aplicaciones clientes mediante el Servicio de Eventos.
Aplicación proveedora	Aplicación generadora de eventos y que los hace llegar a las aplicaciones suscritas mediante el Servicio de Eventos.
Aplicación consumidora	Aplicación que recibe eventos del Servicio de Eventos.
Operador	Operador del Sistema.

Tabla 4.3. Actores.

Para su funcionamiento el Servicio de Eventos, que llamaremos en adelante SE, utiliza un módulo central, que llamaremos en adelante MCSE (Módulo Central del Servicio de Eventos), y envolventes para cada una de las aplicaciones. El MCSE se encarga de recibir y enviar mensajes desde y hacia las diversas envolventes de las aplicaciones que están conectadas a él (*figura 4.4*).

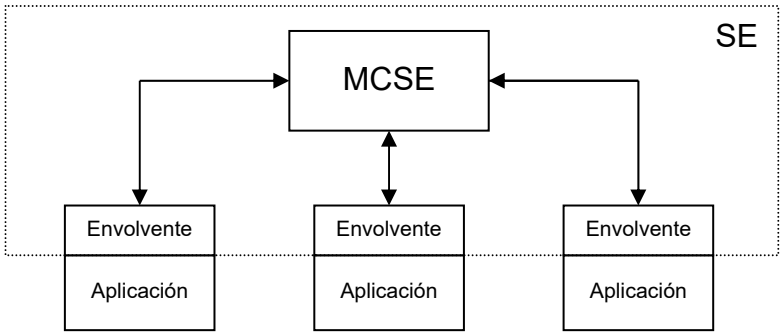


Figura 4.4. Componentes del Servicio de Eventos.

Tipos de mensajes

Existen cuatro tipos de mensajes que se pueden enviar entre las aplicaciones mediante el SE: Solicitud, Solicitud - respuesta, Respuesta y Notificación. En la tabla 4.5 se describe la información que lleva cada uno de ellos.

Tipo de mensaje	Descripción
Solicitud	Información del servicio solicitado y parámetros que requiere el servidor.
Solicitud – respuesta	Información del servicio solicitado y parámetros que requiere el servidor.
Respuesta	Resultado de una solicitud de servicio.
Notificación	Información de la ocurrencia de un evento en una aplicación proveedora.

Tabla 4.5. Mensajes.

Las aplicaciones pueden enviar ciertos tipos de mensajes en función del rol que juegan como se describe a continuación:

- Las aplicaciones clientes envían mensajes de solicitud y de solicitud - respuesta de servicios.
- Las aplicaciones servidoras envían mensajes de respuesta.
- Las aplicaciones proveedoras envían mensajes de notificación.
- Las aplicaciones consumidoras no envían mensajes.

Funcionalidad

Cada una las aplicaciones debe estar registrada en el SE para desempeñar alguno de los roles. Para satisfacer las diferentes necesidades de comunicación se debe contar con la posibilidad de que el cliente inicie una solicitud de servicio o bien que una aplicación que genera un evento se lo comunique a las aplicaciones interesadas.

Los mensajes que se envían al SE deben llevar dos banderas, la primera bandera que se denomina bandera de bloqueo, indica que la aplicación quedará bloqueada en espera de la respuesta; la segunda bandera que se denomina bandera de persistencia, indica que el mensaje requiere entrega garantizada. En la tabla 4.6 se muestran las diferentes combinaciones de las banderas de bloqueo y persistencia en los cuatro tipos de mensajes que maneja el SE.

Tipo de mensaje	Bandera de bloqueo	Bandera de persistencia	Observaciones
Solicitud - respuesta	**1**	**0**	**Con bloqueo, sin persistencia.**
	1	**1**	**Con bloqueo, con persistencia.**
	~~0~~	~~0~~	*Sin bloqueo no se puede esperar respuesta.*
	~~0~~	~~1~~	
Solicitud	~~1~~	~~0~~	*Si se requiere bloqueo, es para esperar la respuesta, por ello debe utilizar el esquema anterior.*
	~~1~~	~~1~~	
	~~0~~	~~0~~	*Las solicitudes sin bloqueo, deben tener entrega garantizada.*
	0	**1**	**Sin bloqueo, con persistencia.**
Respuesta	~~1~~	~~0~~	*Las respuestas son sin bloqueo y deben tener entrega garantizada.*
	~~1~~	~~1~~	
	~~0~~	~~0~~	
	0	**1**	**Sin bloqueo, con persistencia.**
Notificacion	~~1~~	~~0~~	*No se permiten las notificaciones con bloqueo.*
	~~1~~	~~1~~	
	0	**0**	**Sin bloqueo, con persistencia.**
	0	**1**	**Sin bloqueo, con persistencia.**

Tabla 4.6. Comportamiento del servicio de eventos.

No todas las combinaciones de banderas de bloqueo y persistencia son permitidas para los diferentes tipos de mensajes, aquellas que no son validas se encuentran tachadas en la tabla. Las combinaciones válidas se describen a continuación.

a) Solicitud – respuesta de servicio

Con bloqueo, sin persistencia:
La aplicación cliente solicita un servicio y se bloquea hasta recibir una respuesta o bien se cumple un tiempo máximo de espera. El SE determina que aplicación de las que tiene registradas ofrece el servicio, la solicitud es enviada sólo una vez a la aplicación servidora y se espera por una respuesta. Una vez que recibe la respuesta, la aplicación cliente es desbloqueada (*figura 4.7*).

Con bloqueo, con persistencia:

La aplicación cliente solicita un servicio y se bloquea hasta recibir una respuesta o bien se cumple un tiempo máximo de espera. La solicitud es almacenada en una base de datos para poder garantizar la entrega. El SE determina que aplicación de las que tiene registradas ofrece el servicio, la solicitud es enviada el número de veces que sea necesario a la aplicación servidora para obtener la respuesta. Una vez que recibe la respuesta, la aplicación cliente es desbloqueada (*figura 4.7*).

Este tipo de comportamiento es útil cuando una aplicación requiere la respuesta a la solicitud para continuar con su ejecución, sin embargo, si la respuesta no está disponible después de cierto tiempo, ya no le sirve.

Figura 4.7. Solicitud – respuesta con bloqueo.

b) Solicitud de servicio

Sin bloqueo, con persistencia:
La aplicación cliente envía un mensaje de solicitud al SE y es liberada inmediatamente. La solicitud es almacenada en una base de datos para poder garantizar la entrega. El SE envía el mensaje de solicitud a la aplicación servidora el número de veces que sea necesario para garantizar que ha sido recibida (*figura 4.8*).

c) Respuesta de servicio

Sin bloqueo, con persistencia:
La aplicación servidora envía un mensaje de respuesta al SE y es liberada inmediatamente. La solicitud es almacenada en una base de datos para poder garantizar la entrega. El SE envía el mensaje de respuesta a la aplicación cliente el número de veces que sea necesario para garantizar que ha sido recibida (*figura 4.8*).

Este tipo de comportamiento es útil cuando una aplicación puede continuar su ejecución sin la respuesta.

Figura 4.8. Solicitud sin bloqueo, respuesta sin bloqueo.

d) Notificación de evento

Sin bloqueo, sin persistencia:
La aplicación proveedora envía un mensaje de notificación al SE y es liberada inmediatamente. El SE envía el mensaje a las aplicaciones que estén registradas como consumidoras de dicho evento, dicha información la obtiene de la base de datos. El mensaje sólo es enviado una vez las aplicaciones suscritas no importando si todas las aplicaciones la recibieron (*figura 4.9*).

Sin bloqueo, con persistencia:
La aplicación proveedora envía un mensaje de notificación al SE y es liberada inmediatamente. La solicitud es almacenada en una base de datos para poder garantizar la entrega. El SE envía el mensaje a las aplicaciones que estén registradas como consumidoras de dicho evento. Dicha información la obtiene de la base de datos, el mensaje es enviado el número de veces que sea necesario a la aplicaciones suscritas hasta asegurase que todas han sido notificadas (*figura 4.9*).

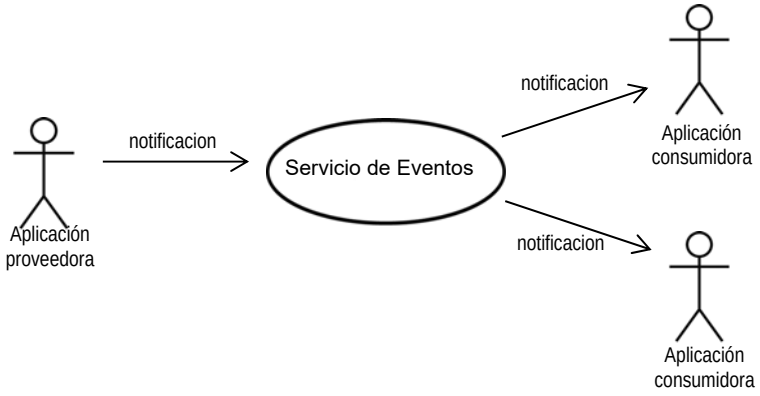


Figura 4.9. Notificación de evento.

Diagramas de Casos de Uso y Diagramas de Actividades

En las tablas de la 4.10 - 4.17 se muestran los Diagramas de Casos de Uso y Diagramas Actividades para el Servicio de Eventos.

a) Registro de la aplicación

Nombre del Caso de Uso	***Registro de la aplicación***
Objetivo	Registrar en el MCSE los datos una aplicación y la forma en que va a interactuar con el SE.
Actores	Operador.
Pre-condiciones	Ninguna.
Flujo Normal	El Operador registra a la aplicación que se integrará al esquema de comunicación. Se deben incluir los datos de la aplicación y el rol que desempeñará.
Flujo Alternos	Ninguno.
Post-condiciones	La aplicación queda registrada en el MCSE. Las aplicaciones son registradas como clientes, servidoras, consumidoras o proveedoras.
Diagrama	
Notas	Todas las aplicaciones deberán estar registradas en el MCSE

Tabla 4.10. Caso de uso "Registro de la aplicación".

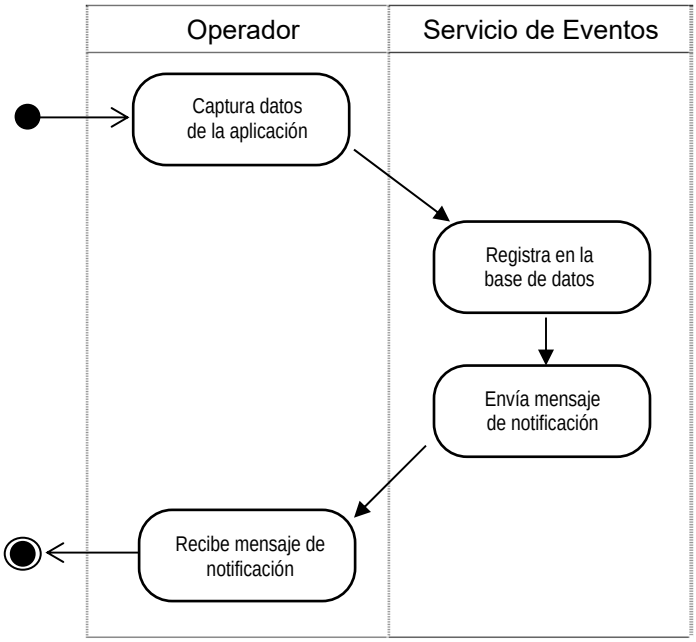


Tabla 4.11. Diagrama de actividades "Registro de la aplicación".

b) Solicitud de servicio

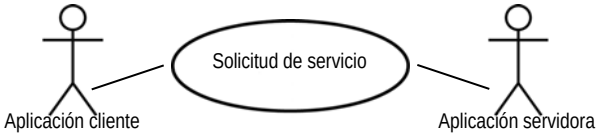
Nombre del Caso de Uso	***Solicitud de servicio***
Objetivo	Enterar a la aplicación servidora que debe proporcionar un servicio a la aplicación cliente.
Actores	Aplicación cliente, Aplicación servidora
Pre-condiciones	Tanto la aplicación cliente como la aplicación servidora deben estar registradas en el MCSE.
Flujo Normal	La aplicación cliente envía un mensaje de solicitud de servicio al SE, la aplicación puede liberarse inmediatamente o quedarse bloqueada en espera de la respuesta. El SE determina que aplicación de las que tiene registradas ofrece el servicio, la solicitud es enviada a la aplicación servidora. El SE recibe la respuesta del servidor y la entrega a la aplicación cliente. En caso de ser una solicitud abanderada con entrega garantizada, el SE la almacena y retransmite el número de veces que sea necesario para garantizar su entrega.
Flujo Alternos	
Post-condiciones	
Diagrama	
Notas	

Tabla 4.12. Caso de uso "Solicitud de servicio".

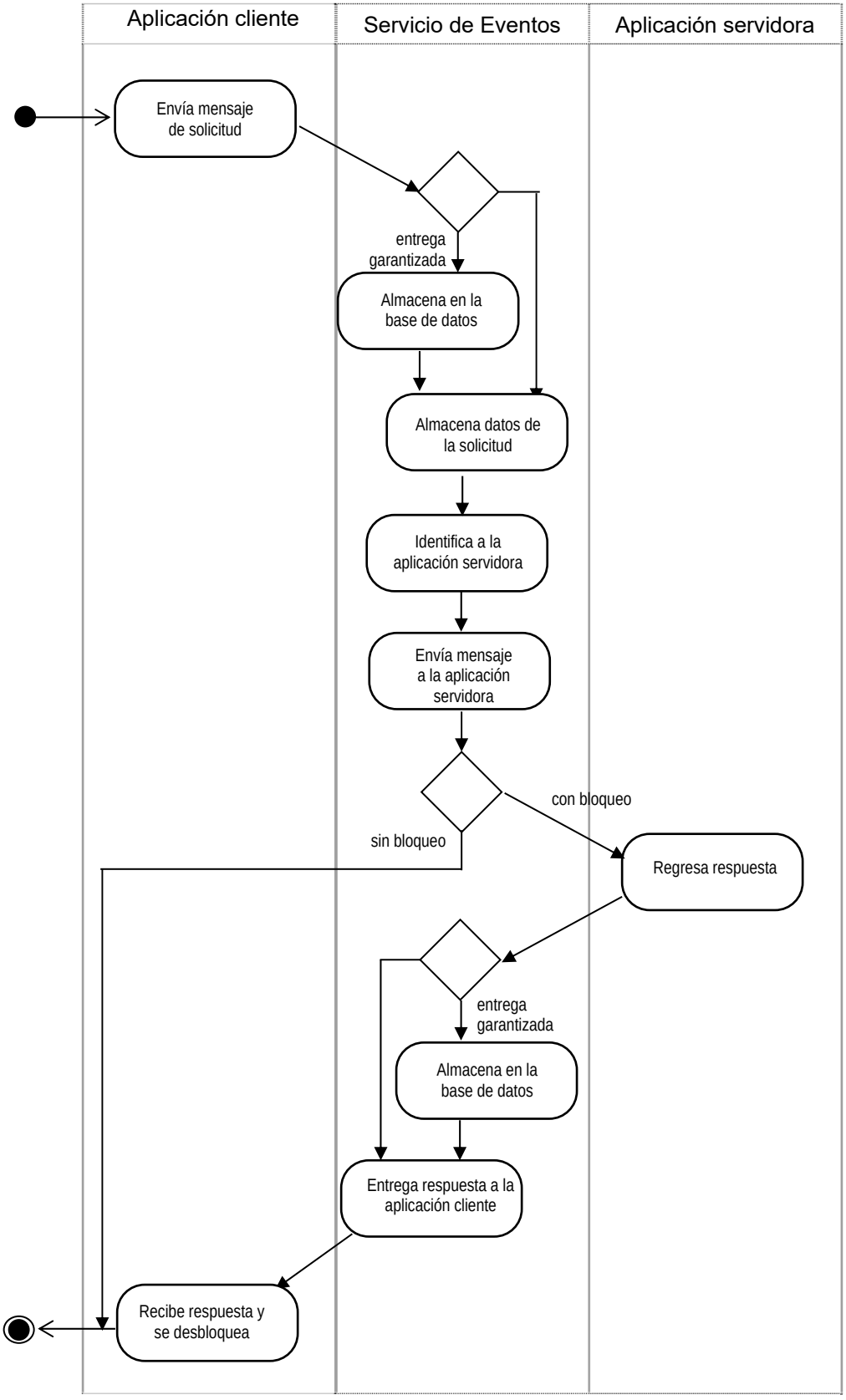


Tabla 4.13. Diagrama de actividades "Solicitud de servicio".

c) Envío de respuesta

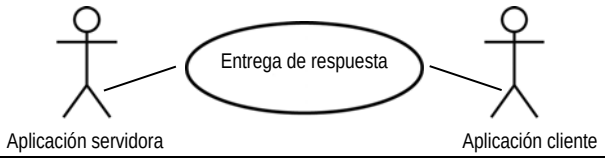
Nombre del Caso de Uso	***Envío de respuesta***
Objetivo	Entregar una respuesta a una aplicación que hizo una solicitud sin bloqueo.
Actores	Aplicación servidora, Aplicación cliente.
Pre-condiciones	Tanto la aplicación servidora como la aplicación cliente deben estar registradas en el MCSE.
Flujo Normal	La aplicación servidora entrega un mensaje de respuesta al SE, éste determina quien está esperando la respuesta y se la entrega. La respuesta se almacena y retransmite el número de veces que sea necesario para garantizar su entrega.
Flujo Alternos	
Post-condiciones	La aplicación cliente recibe la respuesta.
Diagrama	
Notas	Los mensajes de respuesta son sin bloqueo y con persistencia.

Tabla 4.14. Caso de uso "Envío de respuesta".

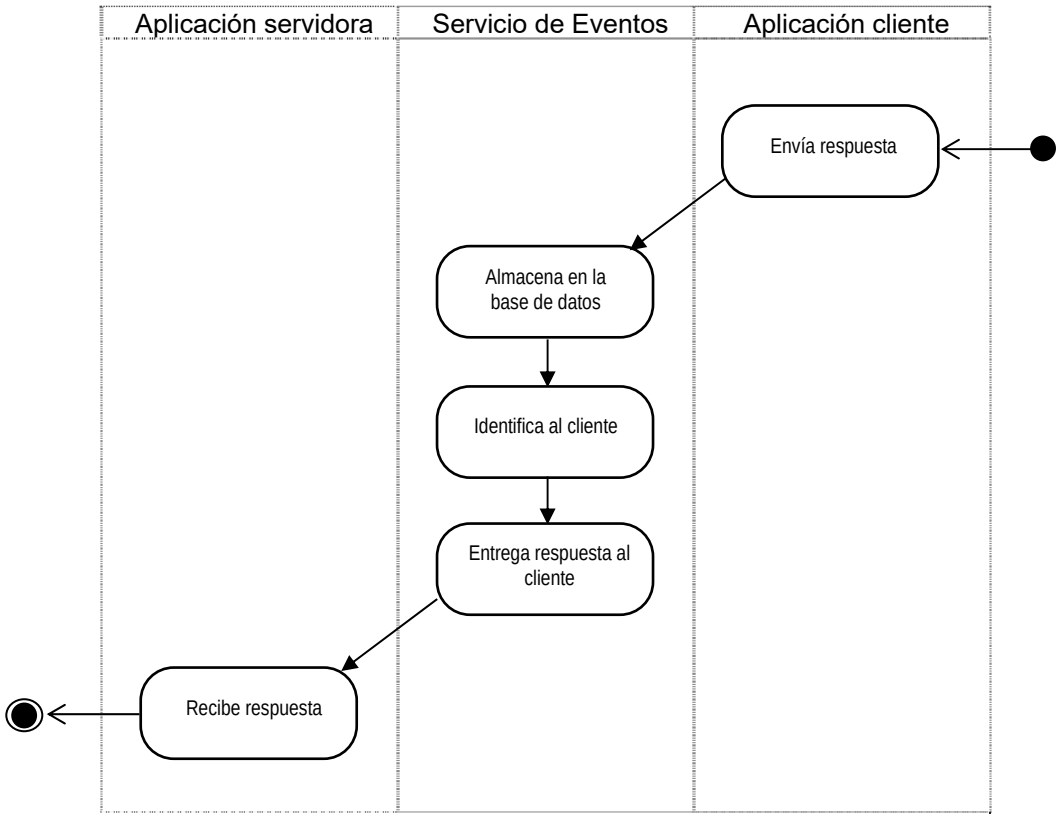


Tabla 4.15. Diagrama de actividades "Envío de respuesta".

d) Publicación de evento

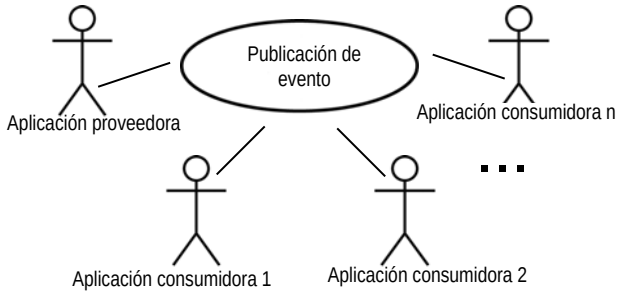
Nombre del Caso de Uso	***Publicación de evento***
Objetivo	La aplicación proveedora notifica al Servidor de Eventos que ha ocurrido un evento.
Actores	Aplicación proveedora, Aplicación consumidora
Pre-condiciones	Tanto la aplicación proveedora como la o las aplicaciones consumidor deben estar registradas en el MCSE.
Flujo Normal	La aplicación proveedora envía un mensaje de notificación al servicio de eventos. El SE envía el mensaje a las aplicaciones que estén registradas como consumidoras de dicho evento. En caso de ser una notificación abanderada con entrega garantizada, el SE la almacena y retransmite el número de veces que sea necesario para garantizar su entrega a todas las aplicaciones suscritas.
Flujo Alternos	
Post-condiciones	Las aplicaciones suscritas al evento son notificadas.
Diagrama	
Notas	Las notificaciones de eventos son mensajes sin bloqueo.

Tabla 4.16. Caso de uso "Publicación de evento".

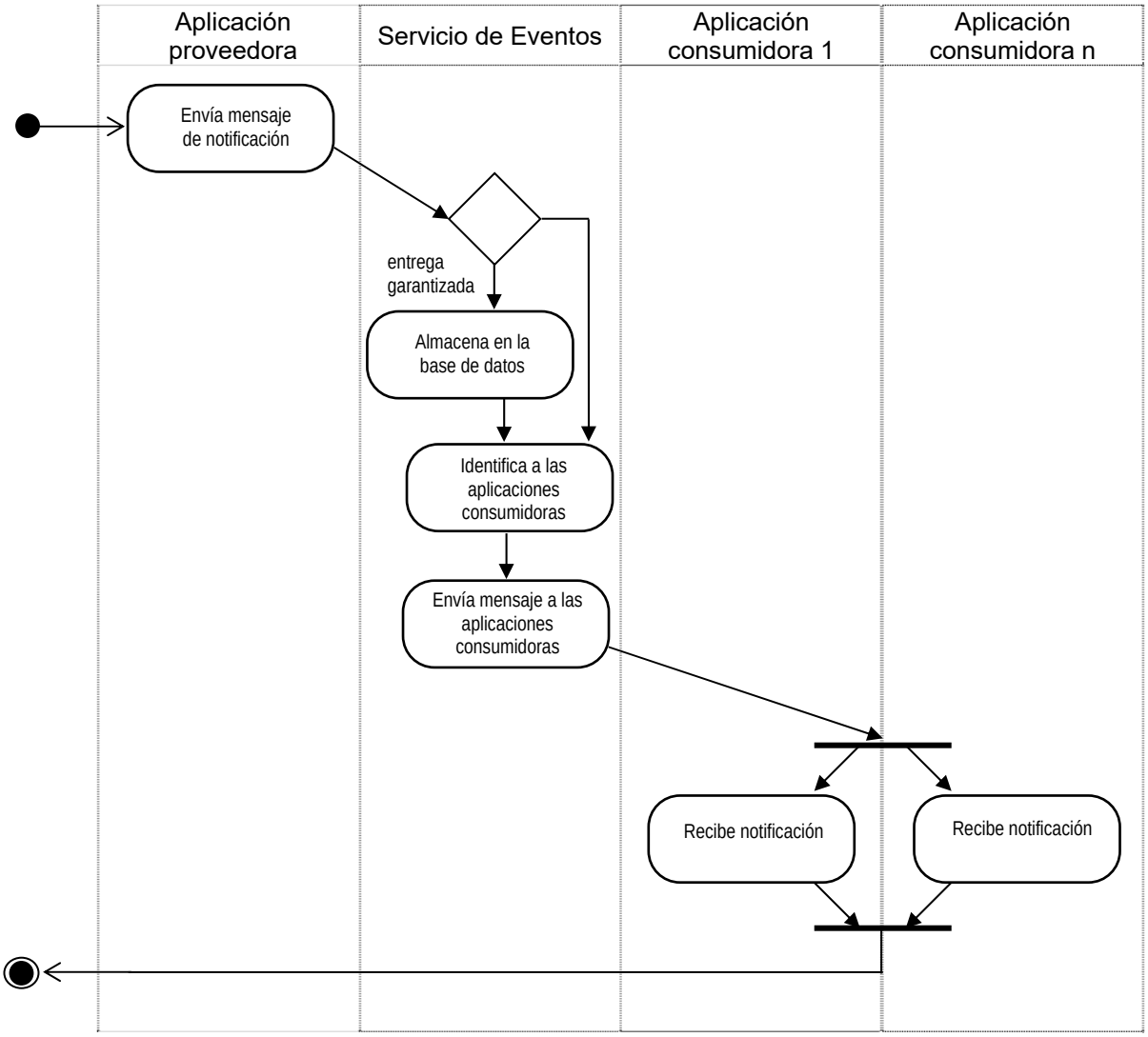


Tabla 4.17. Diagrama de actividades "Publicación de evento".

IV. II Requerimientos técnicos

A continuación se describen los requerimientos técnicos no-funcionales que debe satisfacer el Servicio de Eventos.

Persistencia:Entrega garantizada

El SE debe implementar los mecanismos necesarios para garantizar la entrega de los mensajes que así lo requieran. Debe contar con un dispositivo de almacenamiento persistente que permita almacenar y retransmitir los mensajes el número de veces que sea necesario.

Tolerancia a fallas:Disponibilidad

El SE deberá contar con redundancia en hardware y mecanismos de replicación necesarios que permitan definir un esquema de alta disponibilidad con tolerancia a fallas de hardware. La disponibilidad de un SE en un ambiente de producción, donde interactúa con aplicaciones de misión crítica, debe ser al menos de 99%.

Transparencia en la localización

Tanto los mensajes de solicitud de servicio como los mensajes de notificación de eventos, no deben incluir datos de la localización de las aplicaciones servidoras y consumidoras. El SE debe contar con los mecanismos necesarios para localizarlas.

Manejo de concurrencia

El MCSE debe contar con un adecuado manejo de concurrencia para poder atender los requerimientos de comunicación de las diferentes aplicaciones.

Auditabilidad

El SE debe contar con mecanismos que nos permitan determinar si los mensajes se entregan con éxito o no. En caso de falla en la entrega de un mensaje, nos debe permitir determinar la causa por la que no se entregó el mensaje, ya sea por problemas en la aplicación destino, en la red o en el MCSE.

Flexibilidad:Escalabilidad

La implementación del SE debe permitir la incorporación de aplicaciones sin necesidad de reiniciar el sistema. Si se requiere de más capacidad en el módulo central, se deben poder incorporar más equipos configurados de manera que el sistema sea escalable.

IV. III Diseño del componente de software que implanta el Servicio de Eventos

El diseño incluye la definición de las clases a utilizar para la implantación del SE, así como la estructura y secuencia de los mensajes entre los objetos o instancias de dichas clases.

Diagrama de clases

Para la implementación del MCSE se utilizan dos clases, Registra y Transmite (*figura 4.18*).

- El operador utiliza la clase Registra mediante una interfaz de usuario para registrar las aplicaciones en el MCSE.
- Las envolventes utilizan la clase transmite para hacer llegar los mensajes desde la envolvente que envía hasta la que recibe.

Para la implementación de la envolvente de las aplicaciones se utilizan dos clases, Envía y Recibe (*figura 4.18*).

- Los clientes hacen una llamada a solicita de la clase Envía para solicitar un servicio y los servidores reciben la petición mediante una invocación de sol_servicio de la clase Recibe.
- Los servidores hacen una llamada a responde de la clase Envía para entregar la respuesta y los clientes la reciben mediante una invocación de respuesta de la clase Recibe.
- Los proveedores de eventos utilizan la llamada a notifica de la clase Envía para enviar un evento y a su vez los consumidores lo reciben mediante una llamada de evento de la clase Recibe.

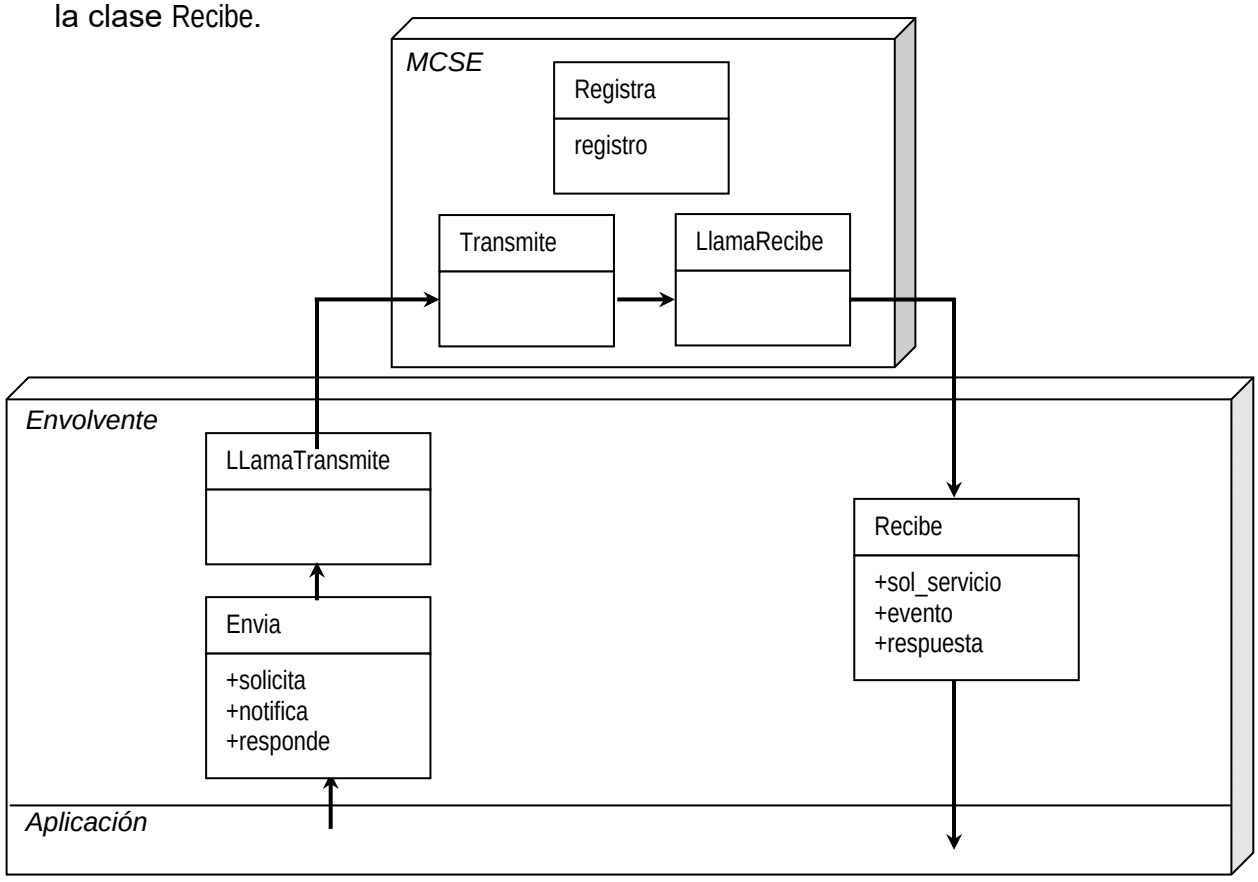


Figura 4.18. Diagrama de clases.

Datos requeridos para el registro de una aplicación

En la tabla 4.19 se muestran los datos requeridos para el registro de una aplicación.

Dato	Tipo	Valores posibles	Descripción
idAplicacion	cadena		Debe incluir la información necesaria para encontrar al equipo y la aplicación.
Rol	entero	1: cliente 2: servidor 3: proveedor 4: consumidor	
eventoServicio	cadena		Servicio que utiliza o proporciona. Evento que publica o al que se suscribe.
listaParametros	cadena		Parámetros de entada y salida para el caso de solicitudes de servicio.

Tabla 4.19. Información de registro.

Estructura de los mensajes

Para los mensajes de solicitud – respuesta y solicitud se define la estructura **m_solicitud.** Incluye las banderas de bloqueo y persistencia. En el caso de que la solicitud sea sin bloqueo, la bandera de persistencia no tiene sentido, ya que las solicitudes sin bloqueo, deben ser persistentes.

```
struct m_solicitud{
     string id;
     boolean b_bloqueo;
     boolean b_persistencia;
     string servicio;
     in string parametros;
     out string parametros;
};
```

Para el mensaje de respuesta se define la estructura **m_respuesta**. No se requieren las banderas de bloqueo y persistencia, ya que las respuestas siempre son sin bloqueo y con persistencia.

```
struct m_respuesta{
     string id;
     string respuesta;
};
```

Para el mensaje de notificación se define la estructura **m_notificacion**. No se requiere la banderas de bloqueo, ya que las notificación son siempre sin bloqueo.

```
struct m_respuesta{
     string id;
     boolean b_persistencia;
     string notificacion;
};
```

Definición de interfaces

```
module Envolvente{
   interface Envia{
      solicita(in m_solicitud mensaje);
      notifica(in m_notificacion mensaje);
      responde(in m_respuesta mensaje);
   };
   interface Recibe{
      sol_servicio(m_solicitud mensaje);
      evento(m_notificacion mensaje);
      respuesta(m_respuesta);
   };
};

module MCSE{
   interface Transmite{
      transmite(in mensaje);
   };
};
```

Diagramas de secuencia

En las figuras 4.20 – 4.23 se muestran los diagramas de secuencia para los diferentes tipos de funcionalidad de transferencia de mensajes que proporciona el Servicio de Eventos.

a) Solicitud - respuesta

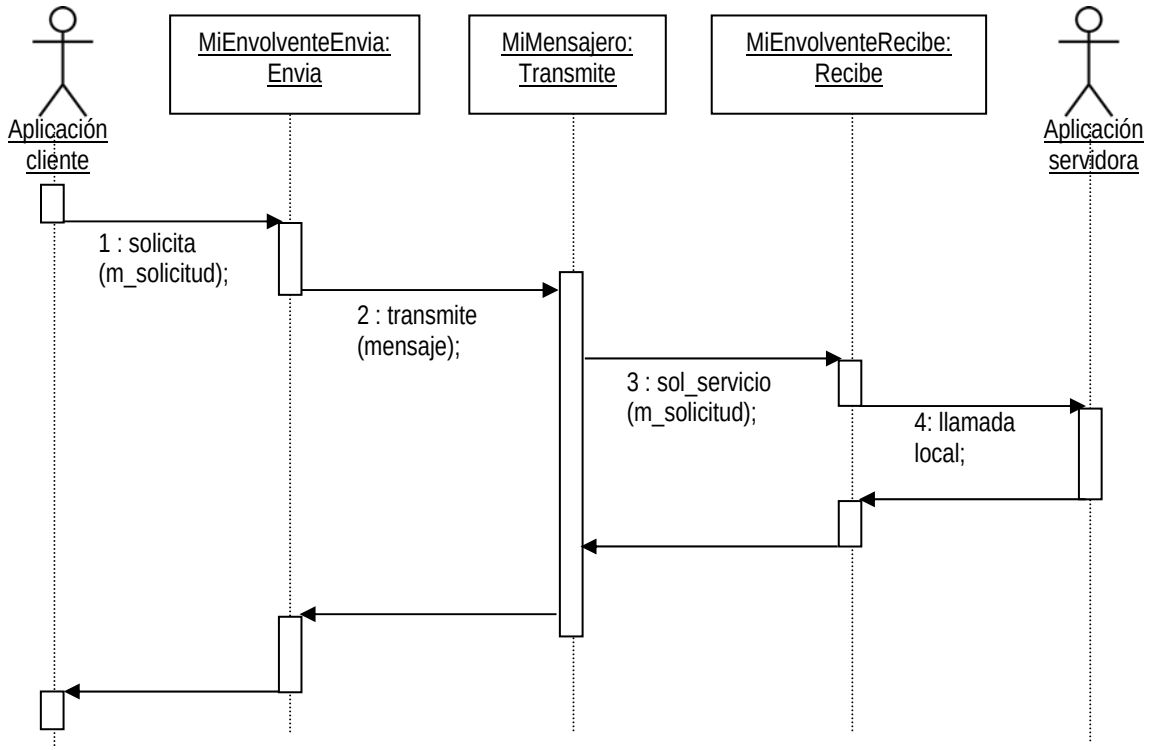


Figura 4.20. Diagrama de secuencia "Solicitud – respuesta".

b) Solicitud

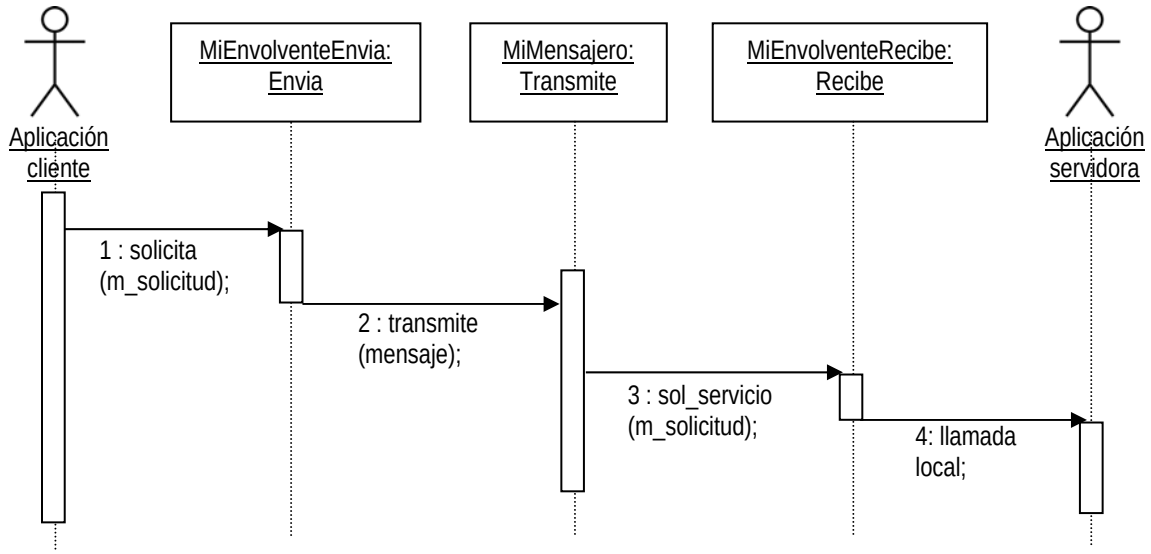


Figura 4.21. Diagrama de secuencia "Solicitud".

c) Respuesta

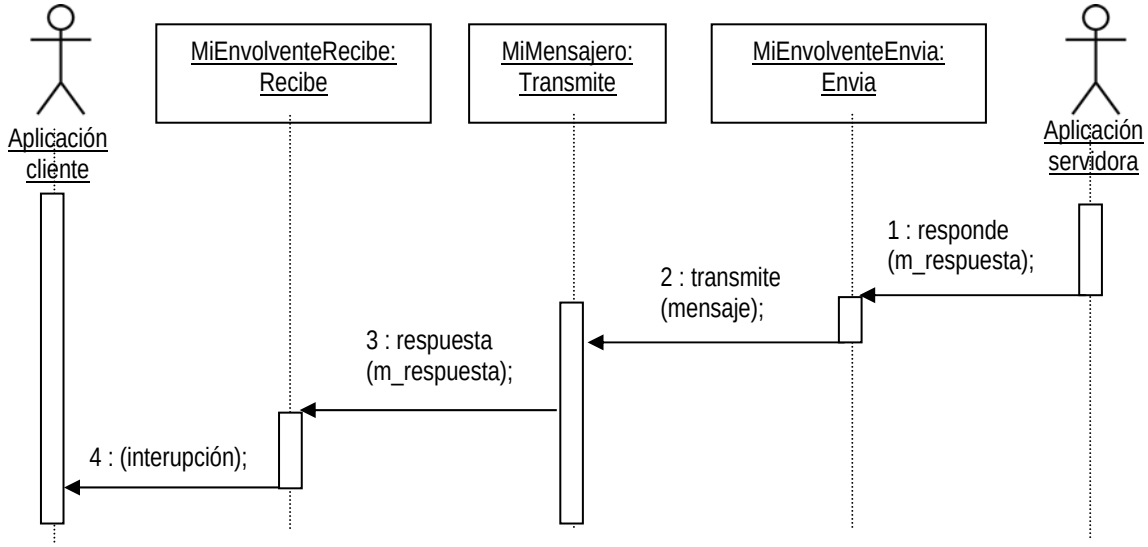


Figura 4.22. Diagrama de secuencia "Respuesta".

d) Notificación

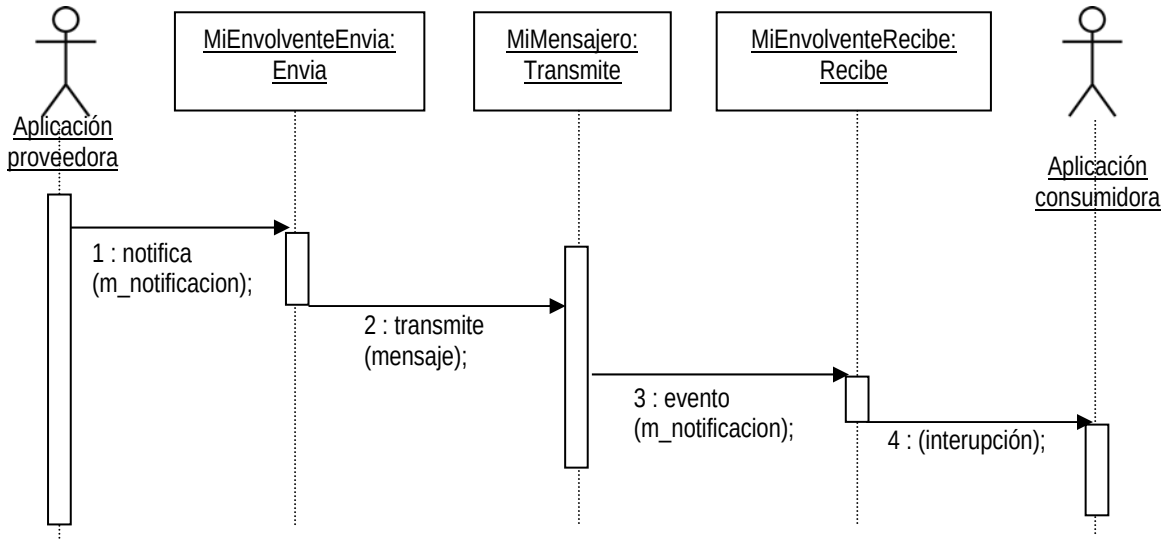


Figura 4.23. Diagrama de secuencia "Respuesta".

IV. IV Alternativas de lenguajes de programación para el desarrollo

El Servicio de Eventos es una sistema para el manejo de mensajes genéricos entre aplicaciones, su diseño es independiente de las aplicaciones con las que estará interactuando. Uno de los retos más importantes es el manejo adecuado de invocaciones remotas de manera transparente entre aplicaciones de diferentes arquitecturas. Durante el diseño del Servicio de Eventos se utilizó IDL (Lenguaje de Definición de Interfaces) para definir las interfaces entre las diferentes clases que implantan el sistema.

Las diferentes tecnologías de desarrollo de aplicaciones distribuidas descritas en el capítulo II utilizan el concepto de IDL para la definición de interfaces, sin embargo CORBA es el único estándar independiente de cualquier tecnología o lenguaje de programación. Existen implementaciones comerciales de CORBA que incluyen mapeadores de IDL a lenguajes orientados a objetos.

Dado que se utilizará una implementación de CORBA para el desarrollo del sistema, es mucho mas natural y sencillo utilizar algún lenguaje orientado a objetos para su desarrollo. Entre los lenguajes orientados a objetos podemos citar: SmallTalk, C++ y Java. A continuación se presenta una descripción de cada uno de ellos con el objetivo de revisar sus características y contar con los elementos para la decisión del lenguaje utilizar.

SmallTalk

SmallTalk es el primer lenguaje de programación puramente orientado a objetos. Cuenta con ambiente de desarrollo que proporciona herramientas que facilitan el uso del lenguaje (editor, explorador, debugger, etc), un compilador y una biblioteca de clases que da acceso al sistema operativo, las interfaces de usuario y otros elementos de diseño de programas.

El ambiente de desarrollo de SmallTalk esta basado en ventanas y botones accesibles con el mouse. Una de las herramientas más utilizadas es el Explorador de la Jerarquía de Clases (CHB), permite explorar, agregar, eliminar y editar las clases utilizadas en un proyecto. Cuenta también con un explorador de disco que permite crear, eliminar, imprimir, renombrar copiar y editar archivos, esta herramienta es particularmente útil para documentar mientras se programa. Proporciona una herramienta de debugger con un inspector que da acceso a las variables de un instancia particular en tiempo de ejecución.

Todo lo que sucede en SmallTalk es resultado de enviar un mensaje a un objeto que soporta un método del mismo nombre que el mensaje. Podemos examinar lo que ocurre a dos niveles: el nivel de envío de mensajes y el de ejecución de métodos.

La sintaxis básica de un mensaje es el destino seguido por el cuerpo del mensaje que puede contener argumentos o no, por ejemplo la siguiente línea de código se interpreta como el mensaje '+' enviado a un objeto 6 con el parámetro 4. Dentro de la filosofía de SmallTalk, los métodos pequeños, fáciles de describir y con un propósito muy específico.

SmallTalk/V incluye más de 100 clases y aproximadamente 2000 métodos (en su versión 286), a medida que se empieza a utilizar, el número de clases y métodos empieza a crecer.

Proporciona una clase **Object** que no puede ser instanciada (es una clase abstracta), se deben utilizar subclases. Uno de las clases más largas es la clase **Collection**. Una colección es una estructura de datos básica utilizada para agrupar comportamiento común entre otras clases. A partir de la clase **Collection** se derivan clases que permiten diferentes formas de organizar sus elementos. Cuenta con para interactuar con el usuario como Dispatcher, Form, Menu, Pane, Prompter, etc.

A pesar de ser un lenguaje puramente orientado a objetos con todas las ventajas que esto ofrece, SmallTalk nunca llegó a ser muy popular. Es muy difícil encontrar aplicaciones desarrolladas en SmallTalk en las empresas, no hay muchos programadores ni soporte.

C++

El lenguaje C es un poderoso lenguaje compilado que ofrece muchas posibilidades a los programadores. Con un diseño cuidadoso, es posible escribir programas en C/C++ portables a la mayoría de las computadoras.

C++ incluye ANSI/ISO C y ofrece la posibilidad de realizar programas orientados a objetos. C es un subconjunto de C++. C fue conocido por mucho tiempo como el lenguaje de desarrollo de UNIX, en la actualidad C/C++ es el lenguaje típicamente utilizado para escribir software de Sistemas Operativos,

Java

Cuando una organización quiere integrarse a internet, Java es el lenguaje más apropiado. Existen varios productos en el mercado para trabajar con Java, sin embargo es suficiente con el *Java Software Development Kit* (JSDK) que puede ser obtenido de la página de Sun Microsystems en Internet. En esta página se encuentra además un excelente tutorial y una documentación completa de las clases de Java.

Otros ambientes de desarrollo de Java (IDE´s): Borland Jbuilder 3, NetBeans DeveloperX2 (gratis para usos no comerciales), Symantec Visual Cafe y Microsoft Visual Java.

Java es un lenguaje completo, permite crear aplicaciones con gráficos, animación, audio y video, multi-hilos, acceso a bases de datos, que pueden correr en Internet y que se pueden comunicar con otras aplicaciones. Todo esto mediante el uso de clases de la biblioteca de Java o mediante la creación de componentes de software reutilizables. Java permite incorporar código en C/C++ en sus programas.

Los programas en Java se componen de clases y estas a su vez de métodos que realizan tareas y entregan información cuando terminan. Se puede programar todo lo que se necesite en Java, sin embargo lo más común es utilizar las clases que proporciona la Biblioteca de Clases de Java. Aprender a utilizar las clases de Java es una parte importante dentro del aprendizaje de Java. Utilizar las clases de Java en lugar de escribir las propias ofrece algunas ventajas, por ejemplo el código está cuidadosamente escrito para tener un buen desempeño y por otro lado las clases de Java están incluidas en prácticamente todas las implementaciones de Java.

El primer paso crear un programa en Java es escribir el código en un editor y salvarlo con una extensión **.java**. El segundo paso es compilar el programa, el compilador de Java traduce el código a byte-code que es el lenguaje interpretado por la máquina virtual de Java. Si el programa compila exitosamente, se genera un archivo **.class** que será utilizado en tiempo de ejecución. Una aplicación es cargada en memoria, verificada y finalmente interpretada mientras se ejecuta. Ya existen compiladores de Java para las plataformas más populares, estos compiladores toman código byte-code o en ocasiones el código fuente y generan lenguaje máquina. Los programas compilados compiten en desempeño con los programas generados por compiladores de C/C++. Existe un paso intermedio entre interpretes y compiladores, JIT (just-in-time) que genera código compilado mientras corre el interprete, de esta manera no tiene que estar re-interpretándose.

Los archivos **.class** pueden contener aplicaciones o applets. Un applet es un programa generalmente pequeño y almacenado en una máquina remota y ejecutado mediante un browser de internet. Los browsers con soporte para Java, cuentan con un interprete de Java y cargan las clases cuando encuentran una referencia a un applet.

Propuesta de lenguajes de desarrollo

Para el MCSE descartamos el uso de SmallTalk ya que no existen mapeadores de IDL para este lenguaje. Tanto para Java como para C++ existen mapeadores de IDL y ambos nos proporcionan las ventajas que ofrece la tecnología orientada a objetos. Si consideramos que el Servicio de Eventos se puede utilizar para solucionar problemas de comunicación en diferentes empresas, es importante que este preparado para residir en cualquier plataforma, por esta razón lo más conveniente es desarrollarlo en Java.

Para las envolventes es necesario contar con versiones para los diferentes lenguajes de programación. Para el caso de Java, la versión es única ya que es multi-plataforma, sin embargo para los lenguajes de programación compilados, incluyendo C++, es necesario crear versiones de la envolvente para cada una de las plataformas de las aplicaciones a integrar.

V IMPLANTACIÓN DEL SERVICIO DE EVENTOS

V. I Utilidad del Servicio de Eventos

A continuación se presenta un ejemplo donde se ve claramente la necesidad de un Servicio de Eventos y como se utiliza para la comunicación de diversas aplicaciones. Se trata de una empresa de Telecomunicaciones que proporciona servicios privados de transmisión de datos. Los servicios pueden tener velocidades de nx64 Kbps, 2 Mbps, 8Mbps y 34.4 Mbps. Para proporcionar el servicio, la empresa debe contar cierta infraestructura de red que está conformada por equipos de transmisión y medios de comunicación.

El diseño de un enlace implica la selección de la ruta a través de los diferentes equipos y medios de tal forma que se pueda proporcionar la velocidad de transmisión entre los sitios solicitados por un cliente. Una vez diseñado el enlace, se asignan los puertos en los diferentes equipos de transmisión para ser activados cuando el servicio sea contratado.

Descripción de la situación inicial

En la figura 5.1 se muestra un diagrama del proceso inicial utilizado por la empresa para el suministro de servicios a los clientes.

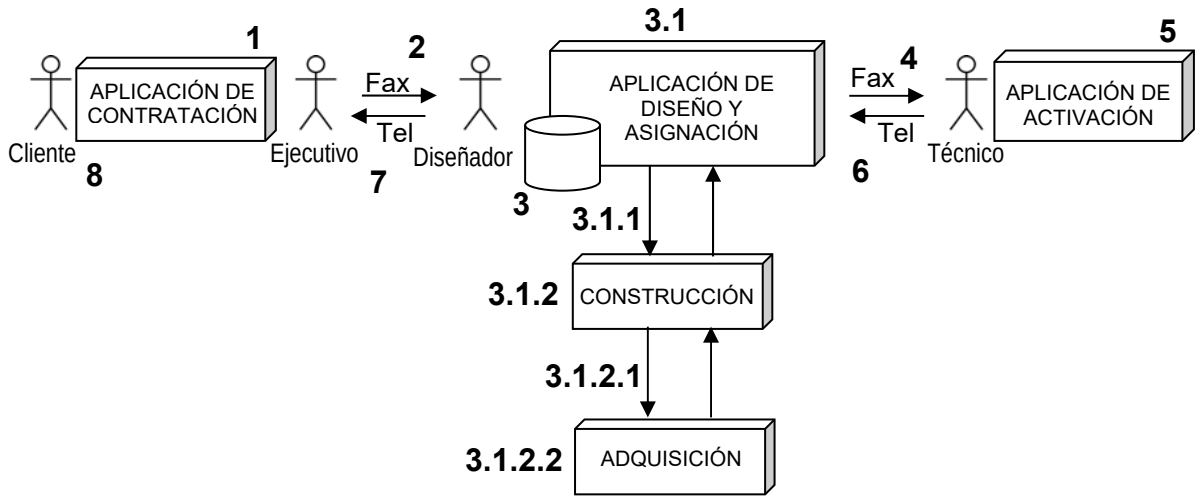


Figura 5.1. Proceso de contratación y entrega de servicios.

1. En la empresa se cuenta con una aplicación que es utilizada por los ejecutivos para la contratación de los servicios. Una vez realizado el contrato, el ejecutivo informa al cliente que su servicio estará listo en un periodo de 20 días hábiles, es decir aproximadamente un mes.
2. El ejecutivo envía un fax al departamento de diseño para solicitar el nuevo servicio, un diseñador recibe el documento y lo atiende para entregar el servicio solicitado.
3. El diseñador cuenta con una aplicación de diseño y activación para desempeñar su trabajo. Primeramente revisa en la base de datos para verificar si existe un enlace previamente diseñado entre los sitios del cliente. Si encuentra uno, lo asigna.
4. El diseñador envía un fax al departamento de activación para solicitar que se configuren y activen los puertos involucrados en el servicio.

5. Un técnico en el departamento de activación realiza las configuraciones solicitadas.
6. Una vez activados los puertos, el técnico notifica al departamento de diseño.
7. El diseñador entrega el servicio activado al ejecutivo.
8. El ejecutivo informa al cliente que su servicio está listo.

Si en el punto 3, el diseñador encontró un enlace diseñado, el servicio se entregó al cliente en dos días.

3.1. En caso de que no haya enlaces ya diseñados y listos para ser activados, el diseñador tiene que hacer el diseño, esto le toma 3 días.
   3.1.1. Si la infraestructura para el diseño no es suficiente, el diseñador hace una solicitud al departamento de construcción.
   3.1.2. El departamento de construcción tarda 5 días para entregar la infraestructura construida.
      3.1.2.1. Si en el departamento de construcción no se cuenta con equipo y medios para instalar, se hace una solicitud al departamento de adquisiciones.
      3.1.2.2. El departamento de adquisiciones tarda un promedios de 10 días en la adquisición del equipo y/o medios solicitados.

En el peor de los casos estamos agregando un total de 18 días al proceso de entrega del servicio (10 para la adquisición, 5 para la construcción y 3 para el diseño), es decir nos tardamos un total de 20 días.

Solución mediante el Servicio de Eventos

Como consecuencia de tener las aplicaciones desintegradas, no podemos tener certidumbre del tiempo que nos tomará entregar el servicio al cliente, el tiempo puede variar desde 2 hasta 20 días. A continuación presentamos la forma de utilizar el Servicio de Eventos para resolver las diferentes necesidades de comunicación entre las aplicaciones involucradas.

Problemática 1. Cuando el ejecutivo esta con el cliente no puede saber si hay servicios listos para ser activados, por lo tanto no sabe si la respuesta de parte del departamento de diseño será en dos o en veinte días.

Integración 1. Desde el sistema de contrataciones se realiza una consulta al sistema de diseño y asignación, de esta manera el ejecutivo puede saber si hay servicios listos para ser activados y darle una información más precisa al cliente.

Para la comunicación entre la aplicación de contratación y la de diseño y asignación se utiliza la alternativa de comunicación **Solicitud – Respuesta** del Servicio de Eventos.

Problemática 2. Cuando hay servicios listos para ser activados, el ejecutivo tiene que enviar un Fax al departamento de diseño y desde esta a su vez se envío otro al departamento de activaciones. Debido al paso manal de información, tardamos dos días en entregar un servicio.

Integración 2. Desde el sistema de contrataciones se envía una solicitud de activación para los casos en los que haya servicios disponibles. Cuando el servicio este activado, el sistema de activaciones envía una respuesta para indicar que el servicio esta listo para ser entregado al cliente. Se reduce el tiempo de entrega de servicios de dos días a unas horas (siempre y cuando no se requieran trabajos en el lado del cliente).

Para la comunicación entre la aplicación de contratación y la de activación se utiliza las alternativas de comunicación **Solicitud** y de **Respuesta** del Servicio de Eventos.

En la figura 5.2 se muestra el diagrama de integración entre la aplicación de contratación y las de diseño y asignación y activación.

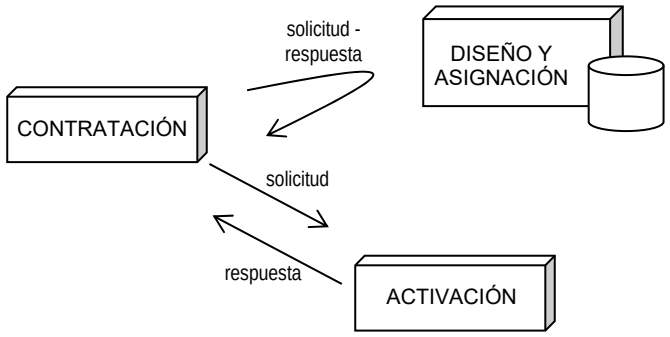


Figura 5.2. Proceso de contratación y entrega de servicios.

Problemática 3. Solo en algunos de los casos de contratación de nuevos servicios, estos ya están diseñados y listos para ser activados.

Para resolver esta problemática es necesario contar con una aplicación para el pronóstico de la demanda. Esta aplicación deberá tomar toda la información de las contrataciones realizadas y hacer análisis estadísticos para determinar donde se deben hacer crecimientos en la red. Una vez construida la infraestructura de red, se actualiza el inventario para que el departamento de diseño pueda hacer uso de esta infraestructura.

Integración 3.1. Desde el sistema de contratación se envía una notificación al nuevo sistema de pronóstico de demanda para que utilice este nuevo dato en su información estadística.

Integración 3.2. Desde el sistemas de pronóstico de la demanda, se hace una notificación al sistema de construcción para que construya la infraestructura de red requerida.
Integración 3.3. Desde el sistemas de pronóstico de la demanda, se hace una notificación al sistema de diseño y asignación para que diseñe los enlaces una vez que la infraestructura está construida.

Para la comunicación entre la aplicación de pronóstico de la demanda y las aplicaciones de contratación, construcción y diseño y asignación, se utiliza la alternativa de comunicación **Notificación** del Servicio de Eventos.

En la figura 5.3 se muestra el diagrama de integración entre la aplicación de contratación, la de pronóstico de la demanda y la de construcción.

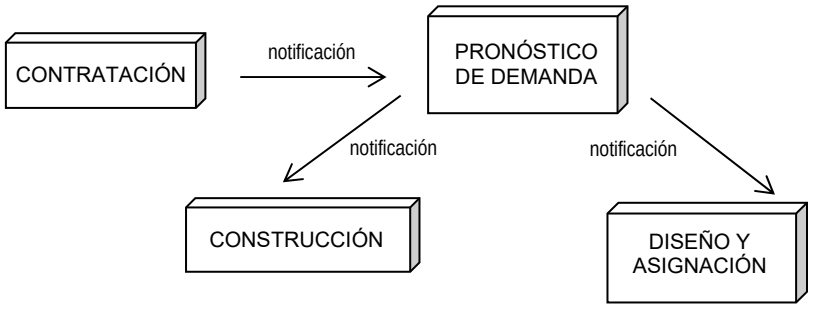


Figura 5.3. Pronóstico de la demanda.

Integración 3.4. Desde el sistemas de construcción se hace una solicitud al sistema de adquisiciones para que se adquieran los equipos y medios necesarios para la construcción de la infraestructura de red.

Para la comunicación entre la aplicación de construcción y la de adquisiciones, se utilizan las alternativas de comunicación de **Solicitud** y de **Respuesta** del Servicio de Eventos.

En la figura 5.4 se muestra el diagrama de integración entre la aplicación de construcción y la de adquisiciones.

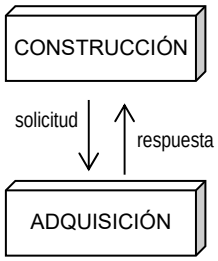


Figura 5.4. Automatización de la compra, construcción y diseño de servicios.

Cada una de las aplicaciones fue creada en diferente época, con equipos de arquitecturas distintas, utilizando diferentes plataformas y con desarrollos en diversos lenguajes de programación. El Servicio de Eventos, al estar basado en CORBA, permite la integración en un ambiente heterogéneo.

Como se puede ver la figura 5.5, la integración mediante el Servicio de Eventos además de habilitar la automatización de procesos, permite eliminar toda relación directa entre aplicaciones. Por ejemplo, cuando el sistema de pronóstico de la demanda hace las notificaciones al sistema de construcción y al de activación, en realidad está interactuando con el Servicio de Eventos. El sistema de pronóstico de la demanda no conoce el detalle de ubicación, tipo de equipo o de aplicación con la que se está comunicando, aún más, no necesita saber si su notificación llega sólo a una sola aplicación o varios.

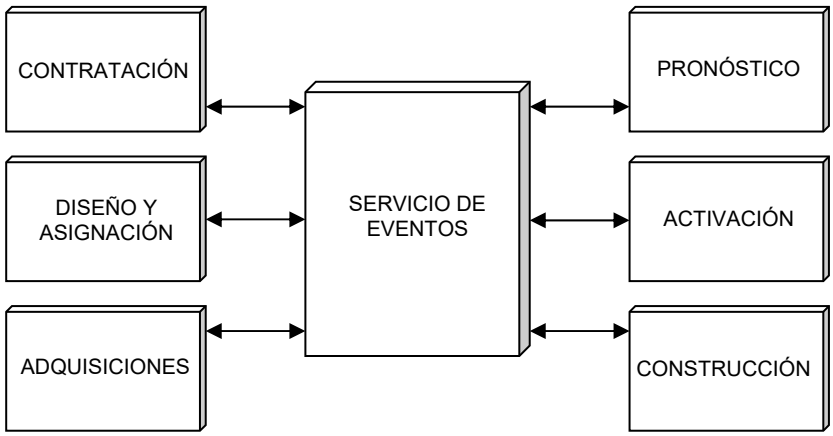


Figura 5.5. Integración mediante el Servicio de Eventos.

Si en algún momento se decide que alguna aplicación requiere cambiar, no es necesario analizar cada una de las interfaces de la aplicación con otros sistemas, simplemente se incorpora el sistema nuevo al Servicio de Eventos, se registran los servicios que proporciona y requiere y se elimina la aplicación antigua.

V. II Implantación

Diagrama general de componentes

Como se muestra en la figura 5.7, para la implantación del MCSE se utiliza un componente que implementa la clase Registra, otro que implementa la clase Transmite, una base de datos de Nombres y otra de Eventos. Para la implantación de las envolventes se utiliza un componente que implanta la clase Envia y otro que implanta la clase Recibe.

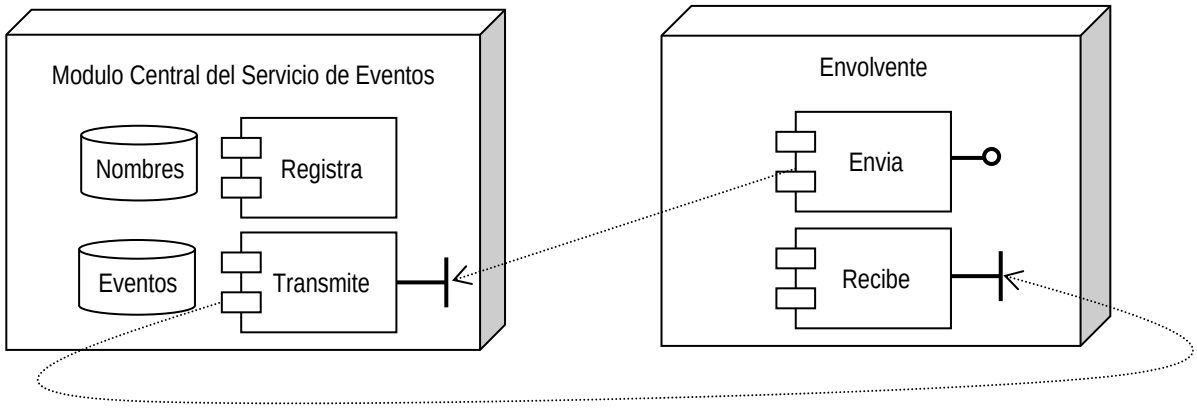


Figura 5.7. Diagrama general de componentes.

El componente Registra es utilizado por el operador del equipo para registrar las aplicaciones que envían y reciben eventos, utiliza la base de datos de Nombres para almacenar la referencia a los objetos.

El componente Transmite recibe todos los eventos las envolventes de las aplicaciones registradas, utiliza la base de datos de Eventos para almacenar los eventos abanderados con persistencia y se comunica con el componente Registra para consultar la referencia de los objetos que deben recibir los eventos.

El componente Envía se utiliza para hacer llegar los mensajes de las aplicaciones hacia el MCSE, el componente Transmite es el que recibe los mensajes y los entrega al componente Recibe en la envolvente.

Especificaciones Técnicas

Manejo de persistencia

Cuando un mensaje esta abanderado con persistencia, el Servicio de Eventos almacena el mensaje en la base de datos de Eventos, cuando el Servicio de Eventos se asegura que todas las aplicaciones lo recibieron, entonces el registro es desechado. Si alguna aplicación no recibió el mensaje, el Servicio de Eventos lo recupera la base de datos e intenta reenviarlo después de un intervalo de tiempo configurable (*figura 5.8*).

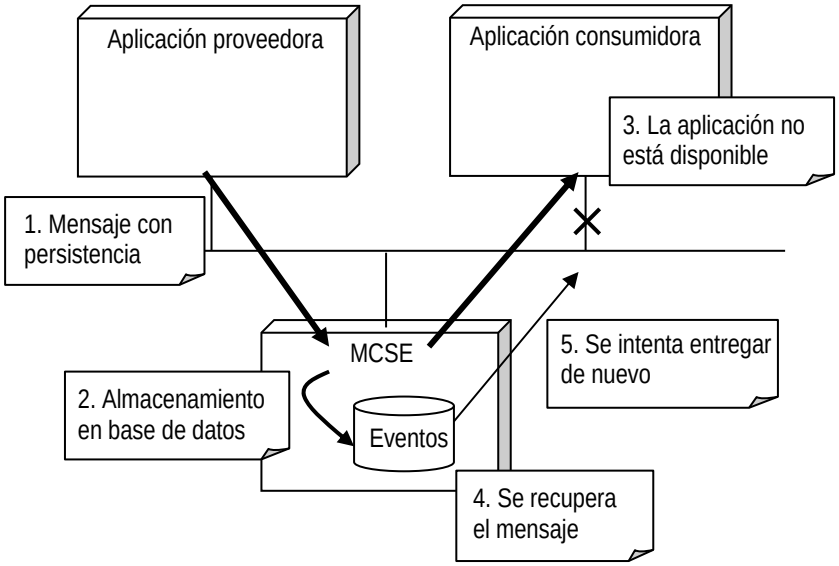


Figura 5.8. Almacenamiento persistente de eventos.

Tolerancia a fallas

Para el manejo de tolerancia a fallas, se utilizan dos equipos para el MCSE (*figura 5.9*).

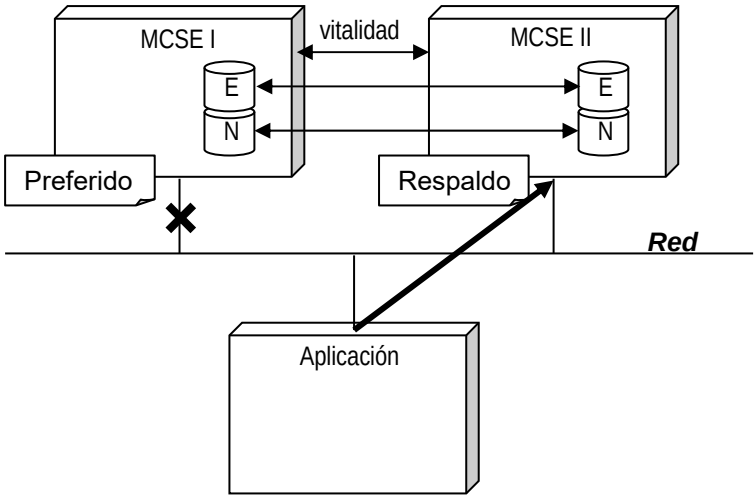


Figura 5.9. Tolerancia a fallas.

Cuando una aplicación se registra, se indica cual de los servidores será el preferido y cual el de respaldo. Las bases de datos de Nombres y Eventos de ambos servidores se mantiene sincronizadas. Cuando el servidor preferido se encuentre fuera de servicio o inalcanzable a través de la red, la envolvente envía los mensajes al servidor de respaldo y éste a su vez a

las aplicaciones interesadas. Una vez que el servidor se recupera, se sincronizan las bases de datos y continúa la operación normal.

Entre los servidores del MCSE se está transmitiendo una señal de vitalidad. Cuando una aplicación envió un mensaje a su servidor preferido, y ocurre una falla antes de que pueda entregarlo, el servidor de respaldo detecta que el otro servidor no está disponible y que hay mensajes sin entregar, entonces el servidor de respaldo se da a la tarea de entregarlos.

Transparencia en la localización

Para mantener transparencia en la localización de las aplicaciones, se utiliza la base datos de Nombres (*figura 5.10*). Cada vez que una nueva aplicación es registrada en el servicio de eventos, los datos del servicio que proporciona o que solicita, así como todos los datos de localización son almacenados en la base de datos de Nombres. Los eventos entre las aplicaciones no contienen datos de localización. Cuando una aplicación solicita un servicio, envía una respuesta o un mensaje de notificación, el Servicio de Eventos recupera los datos de localización de la base de datos y posteriormente transmite el mensaje.

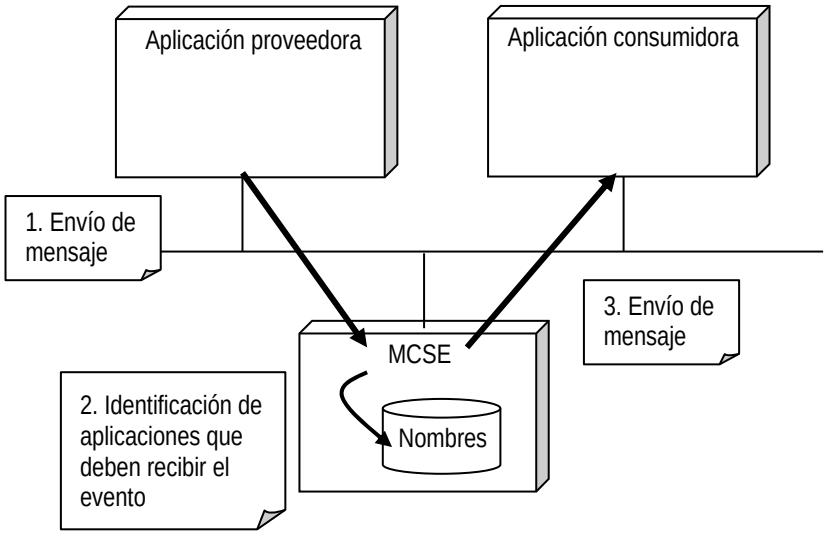


Figura 5.10. Transparencia en la localización.

Si la implementación de alguna de las aplicaciones debe cambiar, en el peor de los casos se deben actualizar los datos de registro. Las aplicaciones con las que interactúa no se enteran que es una nueva aplicación la que proporciona o recibe el mensaje.

Concurrencia

Para que el Servicio de Eventos esté siempre disponible para recibir eventos, se manejan múltiples hilos de ejecución para procesar los mensajes

Auditabilidad

Toda la información concerniente a cada evento es almacenada en un archivo bitácora, accesible mediante una interfaz de usuario (*figura 5.11*).

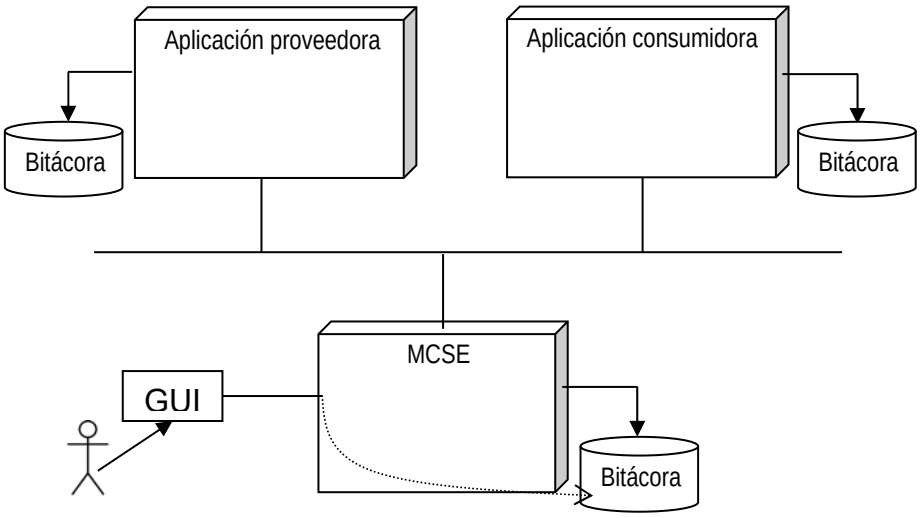


Figura 5.11. Archivos bitácora.

La inserción a dicho archivo bitácora es configurable al igual que la ruta en donde será almacenado dicho archivo.

Escalabilidad

Para agregar nuevas aplicaciones al Servicio de Eventos, se le incorporan las envolventes genéricas. Cuando los recursos de hardware de los MCSE ya no son suficientes, la primera alternativa es crecer el equipo. Si ya no se puede crecer el equipo, se pueden incorporar nuevos equipos para distribuir la carga, al registrar las aplicaciones de debe tomar en cuenta que para cada aplicación debe existir al menos un servidor preferido y otro de respaldo (*figura 5.12*).

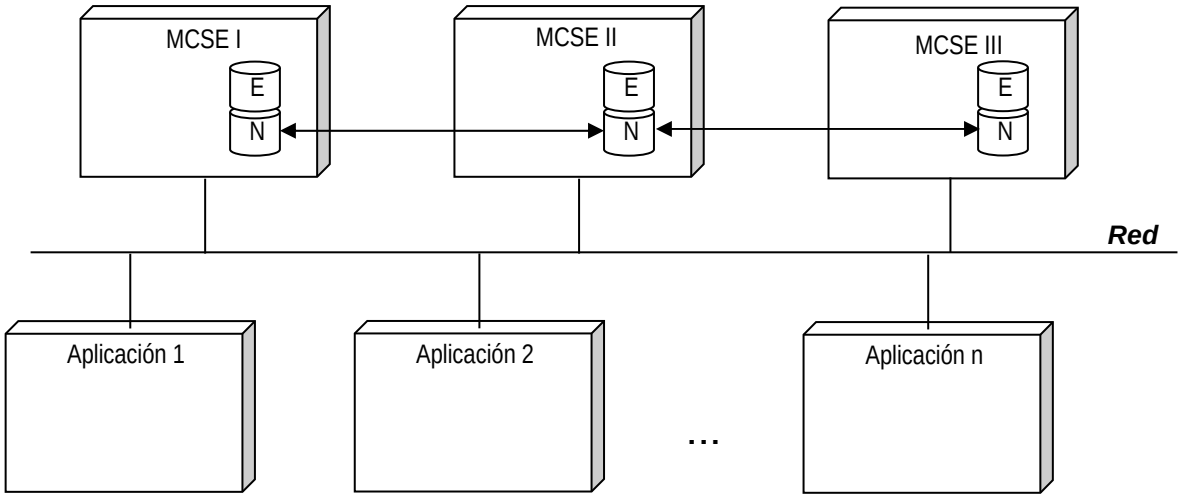


Figura 5.12. Incorporación de nuevas aplicaciones y equipo al MCSE.

Los mensajes siempre están almacenados en la base de datos de dos equipos distintos, el del primario y el de respaldo para cada aplicación. Entre los equipos hay una señal de vitalidad, cuando alguno se cae, el servidor que mantiene el respaldo de dichos mensajes, continúa recibiendo y entregando mensajes.

CONCLUSIONES

Se presentó el Servicio de Eventos como una solución estándar de comunicación en una empresa.

El Servicio de Eventos cuenta con esquemas de comunicación síncrona y asíncrona entre las aplicaciones que permiten el uso de los paradigmas solicitud – respuesta y publicación – suscripción para la integración de aplicaciones, así mismo, permite la comunicación de aplicaciones en un ambiente heterogéneo donde podemos encontrar aplicaciones desarrolladas en diferentes lenguajes de programación y que son ejecutadas en distintas plataformas.

La comunicación entre aplicaciones mediante el Servicio de Eventos no es punto a punto, ya que las aplicaciones envían los mensajes a un Módulo Central que se encarga de distribuirlos. Cuando hablamos de comunicación síncrona, sólo una aplicación debe recibir el mensaje, pero en la comunicación asíncrona, puede haber más de una aplicación interesada en recibir la notificación de un evento.

Cada aplicación cuenta con una envolvente que se encarga de los detalles de la comunicación. Cuando se realizan cambios en una aplicación, ya sea para incorporar una nueva funcionalidad o para modernizarla, no es necesario enterar a las aplicaciones con las que intercambia información.

El Servicio de Eventos permite hacer una clara distinción entre la funcionalidad de la aplicación y la lógica de comunicación, de esta manera el esfuerzo y el costo de mantenimiento de las interfaces es minimo comparado con un ambiente de múltiples interfaces punto a punto.

El Servicio de Eventos cuenta con mecanismos de tolerancia a fallas que permiten minimizar los impactos de las fallas de hardware, software y red. En operación normal, permite enviar mensajes con entrega garantizada, para ello los almacena en una base de datos y no los elimina hasta que todas las aplicaciones destino los hayan recibido.

Para lograr desacoplar las aplicaciones, el Servicio de Eventos trabaja de la mano con un Servicio de Nombres que permite lograr la transparencia en la localización de las aplicaciones que deben recibir los mensajes.

El Servicio de Eventos utiliza el estándar de industria CORBA que especifica la forma de desarrollar aplicaciones distribuidas en ambientes heterogéneos. Para crear una aplicación distribuida se inicia con una interfaz en un lenguaje de definición de interfaces, a partir de esta interfaz se crean varios archivos que son mapeados a algún lenguaje de desarrollo, se agrega la lógica de la aplicación y finalmente se utiliza el compilador del lenguaje seleccionado para generar la aplicación.

El Servicio de Eventos cuenta con una interfaz de administración que permite registrar las aplicaciones y los servicios que proporcionan o que demandan, adicionalmente permite el acceso a las bitácoras para la toma de decisiones. Se puede incorporar una herramienta de

seguimiento a procesos al Servicio de Eventos que permita el manejo de avisos y escalación.

Es importante un adecuado dimensionamiento de recursos de disco duro y RAM en los Módulos Centrales del Servicio de Eventos para evitar problemas de desempeño. Cuando se hayan incorporado un número importante de aplicaciones al esquema de comunicación del Servicio de Eventos, se pueden crecer los recursos del equipo, o bien agregar un equipo adicional.

Una vez resueltas las necesidades de comunicación entre las aplicaciones, es posible la automatización de los procesos de la empresa.

APÉNDICE A: Lenguaje Unificado de Modelado - UML

Un modelo es una representación abstracta de un sistema físico. Las especificaciones UML describen la forma de crear modelos a partir de diagramas y texto. Un modelo tiene un nivel de detalle y un propósito específico dependiendo de la audiencia esperada, mostrando vistas particulares del sistema.

Para poder crear las diferentes vistas, el modelador debe adoptar diferentes roles. Primeramente se modela el entendimiento del problema, los requerimientos, las prácticas y limitaciones, para asegurarse de tener un adecuado entendimiento. Posteriormente se modela la arquitectura, las especificaciones, el diseño, la implantación y el desarrollo. Existen modelos que sirven a los gerentes como apoyo para mostrar los beneficios del proyecto y obtener los recursos necesarios.

UML asume que los modeladores pueden crear sus propios diagramas y modelos, extendiendo así el UML en diversos aspectos para establecer un lenguaje local de modelado.

Elementos de Modelado

Los modelos se realizan a partir de cosas, relaciones, comportamiento e interacciones en un sistema y buscan la forma de organizar esta información. Se distinguen dos tipos de modelos:

- **Modelos Estructurales** - Representan las piezas de un sistema y sus relaciones.
- **Modelos Dinámicos** - Representan el comportamiento de los elementos de un sistema y sus relaciones.

Los elementos de un modelo se nombran de manera única en un contexto (*paquete*) y deben mostrar como pueden ser utilizados (conectados) por otros elementos de modelado. Un elemento importante para manejar los modelos es ocultar cierta información. Las decisiones con relación a la visibilidad pueden ser factores importantes que afecten la organización lógica de los modelos. UML mantiene y extiende la noción de visibilidad a partir de los lenguajes orientados a objetos.

En UML los elementos de modelado pueden ser visibles en tres formas:

- **Público** - El elemento de modelado puede ser visto por cualquier otro elemento de modelado fuera del modelo
- **Protegido** - El elemento de modelado puede ser visto por elementos de modelado de modelos derivados
- **Privado** - El elemento de modelado sólo puede ser visto por elementos de modelado del mismo modelo.

En notación UML, las cosas son expresadas como símbolos o íconos y las relaciones e interacciones son típicamente expresadas utilizando diferentes tipos de líneas. Las líneas no son solo pasivas, tienen un contenido semántico y reglas.

Los elementos de modelado en si no proporcionan tanta información como la la forma en que están conectados. Los modelos no son sólo diagramas, incluyen las especificaciones de los elementos de modelado. Las especificaciones son expresadas en texto y en ocasiones son más importantes que los diagramas.

Diagramas

Los diagramas expresan puntos de vista de un modelo. Un elemento del modelo puede ser presentado en uno o mas diagramas del modelo, es decir un elemento puede ser presentado en diferentes formas en diferentes diagramas, pero debe ser presentado al menos en uno. Los diagramas no tienen una forma especial, pueden estar encerrados en una caja, en un paquete o solos. Existen las siguientes consideraciones:

- La geometría y la geografía no tienen ningún significado en un diagrama UML, es decir, ni el tamaño ni la posición tienen un significado semántico.
- Los diagramas son de dos dimensiones aunque algunas representan formas tri-dimensionales (como cubos)
- Se puede utilizar texto conforme se requiera.

Existen tres tipos de relaciones visibles en los diagramas:

- **Conexión** - Una conexión une íconos y símbolos, formando rutas de conexión. Las conexiones deben estar conectadas en ambos extremos, no se permiten líneas al aire.
- **Contenedores** - Formas cerradas como cajas o círculos que contienen símbolos, íconos y líneas.
- **Uniones visuales** - Elementos cercanos que sugieren cierta relación por su proximidad, por ejemplo un nombre sobre una línea.

Paquetes

Un paquete es un grupo de elementos que puede incluir diagramas, paquetes subordinados y otros elementos de modelado. De acuerdo a la especificación un paquete puede ser utilizado para organizar elementos con cualquier propósito. El rol que juegan los paquetes en el modelado, es similar al de una clase en programación.

Los paquetes pueden estar anidados y pueden hacer referencia a otros paquetes. Proveen las bases para el control de la configuración, el almacenamiento y el control de acceso. Proveen elementos para nombrar a los elementos de modelado. Definen el espacio de nombres para los elementos de modelado. Los paquetes anidados crean una jerarquía de nombres. Los paquetes no pueden ser instanciados. Cada elemento en un paquete debe tener un nombre único, elementos en diferentes paquetes pueden ser llamado de la misma forma, se diferencian por el nombre del paquete como un identificador adicional. Esta capacidad se vuelve significativa en componentes reutilizables y conforme un sistema crece, una colisión de nombres se resuelve poniéndolos en paquetes separados y haciendo referencia a ellos por su nombre completo

nombreDelPaquete::nombredelElemento

Paquetes individuales poseen elementos de modelado, un elemento pertenece únicamente a un paquete, si el paquete es eliminado del modelo, los elementos son eliminados también. Usualmente un paquete puede ser creado, mantenido, actualizado y manejado de forma separada, por eso son considerado en ocasiones como unidades básicas de desarrollo. Los elementos contenidos en el mismo paquete pueden estar relacionados con elementos contenidos en el mismo paquete a diversos niveles.

Los paquetes pueden importar o acceder el contenido de otros paquetes. Cuando un paquete importa otro, importa todos los elementos que son visibles. Los elementos importados pueden ser referenciados por su nombre sencillo y pueden ser usados en relaciones del propio paquete. Los paquetes pueden especializar otros paquetes mediante una relación de generalización. Los paquetes encapsulan los elementos de modelado y definen su visibilidad de la siguiente forma:

- Privado - Los elementos no son disponibles fuera del paquete.
- Protegido - Los elementos son disponibles sólo a paquetes con generalizaciones al paquete que pertenecen los elementos.
- Público - Los elementos pueden ser accesados o importados por otros paquetes.

Los modelos son un tipo de paquetes explícitamente identificado en el meta-modelo UML. Los subsistemas son otro tipo de paquetes explícitamente identificados en el meta-modelo UML. Un subsistema es una forma de representar una unidad de comportamiento en un sistema físico, una forma de particionar el sistema a ser implementado. Es un concepto más físico que lógico o conceptual.

Un modelo puede ser divido en subsistemas, un subsistema puede incluir uno más modelos. Conceptualmente el sistema físico completo es representado a un alto nivel por un simple modelos. El modelo puede entonces ser subdividido según se necesite con propósitos de modelado. Los subsistemas son en si mismos pequeños sistemas que no traslapan funcionalidad. En el proceso de desarrollo los subsistemas proporcionar una forma de organizar los productos de trabajo. Un subsistema hereda nombres, acceso y propiedades de dependencia del paquete. Un subsistema proporciona interfaces y tiene operaciones, su contenido puede ser separado en conjuntos de especializaciones e implementaciones.

Valores etiquetados

Propiedades de elementos explícitamente definidas, expresan información acerca del modelo, o de los elementos de modelado, no del sistema en si. Se definen entre llaves.

{propiedad=”valor”}

Los valores etiquetados pueden ser colocados dentro del contenedor o cerca del elemento a etiquetar. Existen algunos valores etiquetados predefinidos como invariaza, precondición o post-condición.

Constrains

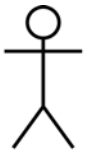
Restricciones semánticas de un elemento de modelado, esencialmente reglas o condiciones semánticas. Al igual que los valores etiquetados, se expresan entre llaves y se colocan cerca del elemento al que hacen referencia.

Símbolos

Los símbolos UML pueden tener contenido o ser sólo íconos. Los íconos son fijos en tamaño y forma, no pueden estar ligados a un conector y pueden estar localizados dentro de los símbolos. Los símbolos bi-dimensionales tiene la capacidad de crecer o tener compartimientos, algunos son gráficas que contienen nodos unidos a conectores.

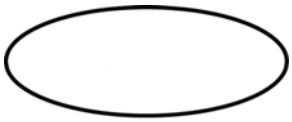
Actor

Entidad fuera del sistema que interactúa directamente con el, típicamente un usuario del sistema u otro sistema o subsistema. Un actor se utiliza en los Diagramas de Casos de Uso y puede ser representado en otro tipo de diagramas del modelo del sistema.



Caso de Uso

Secuencia de transacciones desarrolladas por un sistema en respuesta a un evento iniciado por un actor y que le produce cierto valor.

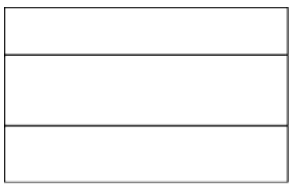


Colaboración

Colección de objetos que interactúan para implementar un comportamiento. Típicamente son utilizados para especificar la implementación de un caso de uso o una operación. También puede ser utilizado para expresar un patrón de software y con un diagrama parametrizado de colaboración, con participantes abstractos, se puede especificar un patrón de arquitectura.

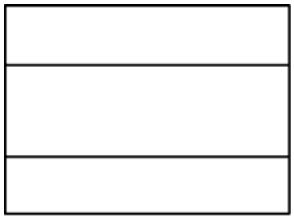
Clase

Una clase es una abstracción de un conjunto posible de objetos que comparten los mismos atributos, operaciones, métodos, relaciones y semántica. Una clase puede utilizar un conjunto de interfaces para especificar la colección de operaciones que provee a su entorno.



Tipo

Un tipo es una colección de objetos que no especifican la implementación física como una clase, los tipos y las clases utilizan el mismo símbolo en UML.
Objeto: Un objeto es una instancia de una clase o un ejemplo de un tipo. Se utiliza el mismo símbolo que para una clase, pero con el nombre subrayado.



Clase activa

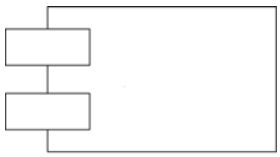
Una clase activa es un conjunto de objetos, cada uno de los cuales posee su propio hilo de control, es decir puede iniciar, controlar y terminar un proceso o un hilo de ejecución.

Interfaz

Una interfaz describe las operaciones visibles de una clase, un componente o un paquete. Define los servicios que son disponibles del elemento implementado.

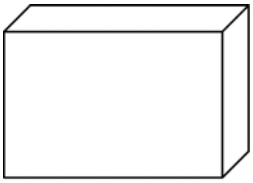
Componente

Parte física, re-localizable de un sistema que empaca la implementación y provee la realización de un conjunto de interfaces. Representa una pieza física de implementación de un sistema, incluyendo código (fuente, binario o ejecutable) o equivalentes tales como scripts o archivos de comandos.



Nodo

Representa un recurso de procesamiento que existe en tiempo de ejecución, con al menos memoria y típicamente capacidad de procesamiento. Incluyen dispositivos de cómputo y otros recursos físicos utilizados en un sistema tales como personas o máquinas.

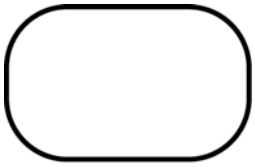


Paquete

Es un contenedor UML utilizado para organizar los elementos de modelado.

Estado

Es una condición, status o situación de un objeto como parte de su ciclo de vida y/o como resultado de una interacción.



Nota

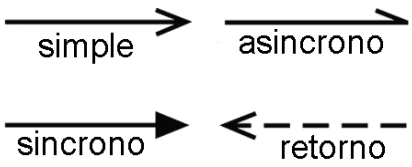
Son utilizados para proporcionar texto explicatorio, tales como comentarios al modelo. Puede ser utilizado en diferentes formas dependiendo de la forma de trabajo del modelador.

Líneas

En UML son utilizadas para expresar mensajes, conexiones dinámicas entre elementos de modelado, ligas e interacciones. Generalmente los mensajes no aparecen en los modelos estructurales y las ligas no aparecen en modelos dinámicos.

Mensajes

Son utilizados en interacciones entre elementos de modelado en los modelos dinámicos, aquellos que representan el comportamiento de un sistema. Los mensajes llevan información entre los objetos. Hay cuatro tipos de mensajes:



- Mensaje simple - El control pasa de un objeto a otro sin proporcionar detalles.
- Mensaje síncrono - El objeto que envía se detiene para esperar el resultado.
- Mensaje asíncrono - El objeto que envía no se detiene para esperar por el resultado.
- Mensaje de retorno - Indica un retorno de una llamada a un procedimiento.
- Relaciones - son utilizadas en modelos estructurales para mostrar las conexiones semánticas entre los elementos del modelo.

Relación de Dependencia ("using")

La conexión entre dos significa que si una cambia, afecta a la otra. Las relaciones de dependencia pueden ser utilizadas para identificar la conexión entre una variedad de elementos de modelado. Son relaciones unidireccionales.

Relación de Generalización

Relación entre dos elementos en la cual uno es una forma más general del otro. La herencia entre clases es representada en esta forma, aunque puede ser utilizada en forma más genéricas, por ejemplo entre paquetes.

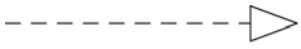
Relación de Asociación ("estructural")

Mapean un objeto a otro conjunto de objetos. Son utilizadas para identificar la ruta de comunicación entre un actor y un caso de uso.

UML describe las relaciones de asociación como el pegamento de los elementos de un sistema, tienen un sentido de navegación, es decir, como podemos llegar de un objeto a otro.

Relación de Realización

es un tipo de relación de dependencia que identifica una liga contractual entre elementos, por ejemplo una clase implementa el comportamiento en un elemento especifico, en este caso una interfaz. Son utilizadas también en casos de uso y diagramas de colaboración.



Relación de Asociación Calificada

Asociación plana con un indicador de la información a utilizar para identificar el objeto destino en un conjunto de objetos asociados.

Relación de Agregación

Representa la parte-completa de una relación. En contraste con una asociación plana que muestra una relación entre iguales, dependiendo del número. En este tipo de relación un elemento es el todo y el otro u otros son las partes.

Relación de Composición

Es una agregación en la que si el elemento que representa el todo desaparece, las partes desaparecen también.

Relación de Dependencia

Las relaciones de dependencia son frecuentemente estereotipadas para soportar las necesidades específicas de un tipo de diagrama.

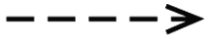
- Extends – Para indicar comportamiento opcional en un caso de uso.

- Includes – Para indicar comportamiento que es común en varios casos de uso.

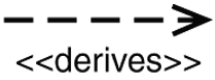
- Imports – Indica dependencia entre paquetes. El paquete puede acceder los elementos con visibilidad pública de los paquetes importados.

- Refines – Indica dependencia entre dos elementos a diferentes niveles de abstracción.

- Realizes – Utilizada para modelar refinamientos de una etapa, la optimizaciones, transformaciones, síntesis, etc.



- Derives – Indica que la instancia de un elemento puede ser obtenida a partir de la relación con otro elemento.



- Trace – Muestra la conexión entre elementos de modelado o conjuntos de elementos de modelado representando el mismo concepto en diferentes modelos.

Diagramas

Los diagramas se forman al colocar los elementos de modelado juntos. No existe una forma de delimitar el alcance de los diagramas y no existe relación entre ellos. Los diagramas son presentaciones gráficas de un modelo. Están acompañados de texto descriptivo que provee la especificación del modelo.

La especificación UML incluye nueve tipos de diagramas diferentes, los diagramas son sugerencias y se utilizan según se requieran en función del proceso utilizado.

UML permite al modelador combinar cualquiera de los elementos de modelado en los diagramas dependiendo de las necesidades de modelado. En la práctica, sólo ciertas combinaciones tienen sentido.

Diagrama de clases

Los diagramas de clase son fundamentales en el modelo de un sistema. Es una vista estructural y estática del modelo. No es una partición formal del modelo. Puede contener interfaces, paquetes, relaciones, objetos y ligas.

Diagrama de Caso de Uso

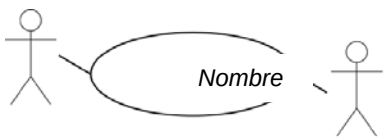
Los diagramas de casos de uso sirven para especificar la funcionalidad y el comportamiento de un sistema mediante su interacción con los usuarios y/o otros sistemas. O lo que es igual, un diagrama que muestra la relación entre los actores y los casos de uso en un sistema.

Para un desarrollo basado en casos, los casos de uso son fundamentales en un esfuerzo de modelado. Los diagramas de casos de uso, muestran los actores, los casos de uso y sus relaciones. La relaciones utilizadas son las siguientes:

- **Asociación** – entre actores y casos de uso.
- **Generalizaciones** – entre actores.
- **Generalizaciones, Extensiones “extends”** e **Inclusiones “includes”** - entre casos de uso.

Captura la parte significativa de la interacción del sistema o parte del sistema y los actores, definiendo el alcance, contexto y requerimientos. Pueden ser utilizados en una variedad de formas durante el esfuerzo de desarrollo, por ejemplo para definir ambientes de pruebas o como un punto de partida para un manual de usuario.

Los diagramas de casos de uso pueden ser encerrados en un rectángulo para indicar el sistema que los contiene. Los diagramas de Casos de Uso son triviales y la verdadera esencia esta en el texto que los acompaña.

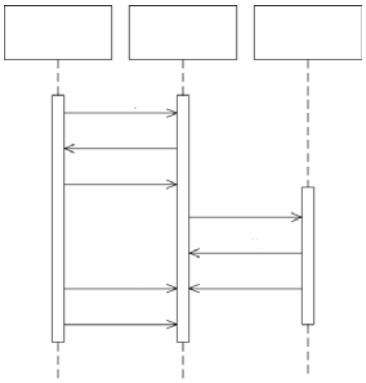


- **Objetivo -**
- **Actores -**
- **Flujo normal -**
- **Flujos alternos -**

Diagrama de Interacción

Los diagramas de interacción son utilizados en el modelado dinámico de un sistema. Existen dos tipo de diagramas:

- **Diagramas de Secuencia** – Muestra la interacción ordenada en una secuencia de tiempo, muestra los objetos y los mensajes que pasan entre ellos cuando ocurre una interacción. También conocidos como diagramas de interacción. Tiene una lista de objetos participantes en rectángulas a lo largo de la parte superior. Cada rectángulo-objeto tiene al menos un nombre, siempre subrayado para indicar que es un objeto y no una clase. Abajo de cada rectángulo se coloca una línea punteada que indica el ciclo de vida de un objeto, que es e marco de referencia para el intercambio de mensajes entre los objetos. Un rectángulo angosto vertical llamado activación, representa el periodo de tiempo que un objeto esta desempeñando una acción ya sea directamente o a través de un intermediario. Los mensajes entre objetos se muestran como flechas con un texto descriptivo.



- **Diagramas de Colaboración** – También muestra el paso de mensajes entre objetos, enfocándose en los objetos y mensajes y su orden en lugar de la secuencia de tiempo. La secuencia de interacciones y los hilos de ejecución concurrentes se identifican con una secuencia de números. Muestra una interacción organizada al rededor de los roles en la interacción y sus ligas con otros, muestra la relación entre objetos jugando diferentes roles.

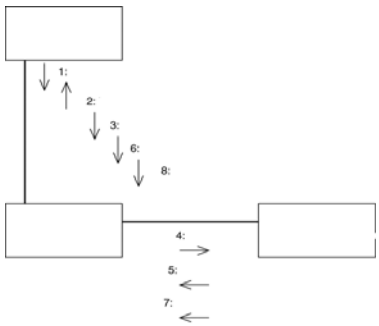


Diagrama de estados

Muestra los estados, eventos, transiciones y actividades del sistema, son parte del modelo dinámico. Utilizados para describir el comportamiento de un elemento de modelado tal como un objeto o una interacción, describe la posible secuencia de estados y acciones a través de las cuales pasan los objetos durante su ciclo de vida como resultado de la reacción a eventos discretos tales como señales o llamados a operaciones.

Un diagrama de estados es típicamente utilizado para modelar el comportamiento de un objeto que requiere una descripción completa de sus estados discretos. Muchos objetos en sistemas estándares de negocio pueden no tener estados significativos, en este caso no requieren un diagrama de estados. Se asociaría típicamente con una y sólo una clase y lista todos los estados que un objeto particular puede tener durante la operación del sistema.

Cada rectángulo redondeado representa un estado en el un objeto puede estar. Un diagrama de estados debe representar todos los estados en que un objeto puede estar y definir un estado inicial que se representa con un circulo relleno. Puede ser que un sistema no tenga un estado final como en el caso de que haya sistemas que deban siempre corriendo. Un objeto pasa de un estado a otro siguiendo una transición disparada por un evento dentro del sistema, típicamente un invocación a un método. Una transición se representa con un flecha de un estado a otro y se asocia al evento que la dispara.

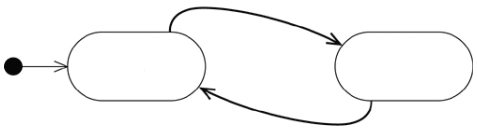


Diagrama de actividades

Un diagrama de actividades es un caso especial, de un diagrama de estados en el cual todos o la mayoría de los estados, acciones o estados de subactividades y todas o la mayoría de las transiciones son disparadas por el final de las acciones o subactividades en los estado origen. Son versiones sofisticadas de diagramas de flujo utilizados en flujos de trabajo y procesos. Un diagrama de actividades esta asociado a una clase y su caso de uso, a un paquete o a la implementación de una operación. Se enfocan en los flujos manejados por procesos internos mas que a eventos externos. Se recomienda su uso cuando los eventos involucrados representan completamente las acciones generadas internamente.

En un diagrama de actividades se utilizan carriles como delimitadores de la responsabilidad de cada actividad. Cuando es necesario indicar que dos o más acciones ocurren en paralelo, se utilizan barras de sincronización.

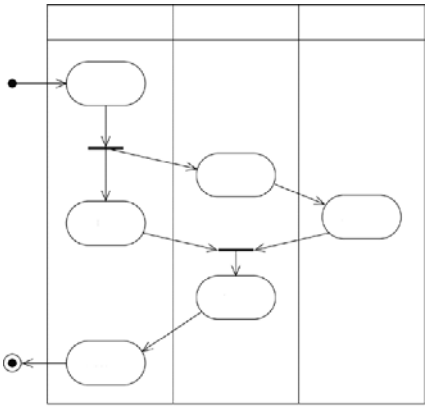
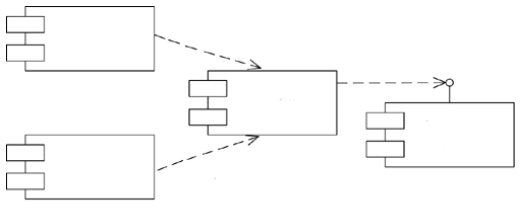


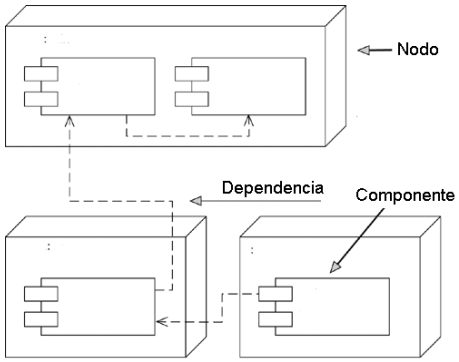
Diagrama de implementación

Modelan los elementos de implementación y consideraciones, incluyendo como el código fuente es estructurado y como y donde están disponibles los ejecutables. Pueden ser utilizados en sentido amplio. Los componentes documentos y procedimientos de negocio y la estructura en tiempo de ejecución la forman las unidades de organización y recursos de negocio incluyendo recursos humanos. Hay dos tipos de diagramas de implementación.

- **Diagrama de componentes** – Muestra la estructura del código. Muestran las dependencias en el código, incluyendo componentes de código fuente, componentes de código binario y componentes ejecutables



- **Diagrama de distribución “deployment”** – Muestra la estructura del sistema en tiempo de ejecución, incluyendo configuración de elementos de procesamiento en tiempo de ejecución y de los componentes de software, procesos y objetos. Para el modelado de negocio, los elementos en tiempo de ejecución incluyen trabajadores y unidades organizacionales, y los componentes de software incluyen procedimientos y documentos utilizados por los trabajadores y unidades organizacionales.



APÉNDICE B: Código de programas de comunicación

A continuación se muestran ejemplos de código de programas utilizados para la comunicación entre aplicaciones para diferentes soluciones como son: Sockets, RPC, RMI, y CORBA.

1. Sockets

En el siguiente ejemplo se muestra el código en C utilizado para recibir mensajes por un servidor. Los archivos i_tcp_se.c e i_tcp_cli.c son los archivos del servidor y del cliente respectivamente y los archivos i_adrr.h y scomun.h son archivos de cabecera. En los recuadros grises se incluye la definición de estructuras y prototipos de funciones incluidos en los archivos de cabecera del sistema.

```
/***********
ARCHIVO: i_tcp_se.c
Archivo tipo para un servidor con conectores AF_INET, SOCK_STREAM.
***********/

# include “scomun.h”
# include “i_addr.h”

main(int argc, char *argv [ ]){
     int sfd, nsfd, pid;
     struct sockaddr_in ser_addr, cli_addr;
```

Definición de la estructura **sockaddr_in** en <sys/netinet.h>

```
struct in_addr {
        u_long          s_addr;
};
struct sockaddr_in {
        short           sin_family; /* AF_INET */
        ushort          sin_port; /* Identifica el proceso */
        struct in_addr  sin_addr; /* Dirección origen o destino */
        char            sin_zero [8];
};
```

```
     nombre_prog = argv [0];

     if((sfd = socket(AF_INET, SOCK_STREAM, 0)) == -1 {
          ERROR (dep, “ABRIR SOCKET”);
          EXIT (-1);
     }
```

Definición de la llamada socket( )
int socket (int af, int type, int protocol);

```
ser_addr.sin_family = AF_INET;
ser_addr.sin_addr.s_addr =
     inet_addr(DIRECCION_NODO_SERVIDOR);
ser_addr.sin_port = htons(PUERTO_SERVIDOR_TCP);
if(bind(sfd, &ser_addr, sizeof(ser_addr)) == -1){
     error(DEP, “bind”);
     exit(-1);
}
```

Definición de la llamada bind( )
int bind (int sfd, const void *addr, int addrlen);

```
listen(sfd, 5);
```

Definición de la llamada listen( )
int listen (int sfd, int backlog);

La llamada listen( ) habilita una cola asociada la conector sfd.
En esta cola se alojan las peticiones de los procesos clientes.

```
for(;;){
     cli_addr_len = size_of(cli_addr);
     if((nsfd = accept(
          sfd, &cli_addr, &cli_addr_len)) == -1){
          error(DEP, “bind”);
          exit(-1);
     }
```

Definición de la llamada accept( )

int accept (int sfd, const void *addr, int *addrlen);

Extrae una petición de la cola creada po la llamada listen( )
¿Crea un nuevo conector, nsfd.

La cola esta vacía:
(0_NDELAY) = desactivo::(bloqueante)
queda en espera de recibir nuevas peticiones de conexión.
(0_NDELAY) = activo::(no bloqueante)
regresa un error.

```
          /*Creación del proceso hijo para atender al cliente*/ if
     ((pid = fork ()) == -1)
               error (DEP, “fork”);
          else if (pid == 0){
               /*Código del proceso hijo*/
               close (sfd);
               recibir_mensaje_str (nsfd);
               close (nsfd);
               exit (0);
          }
          /*Código del proceso padre*/
          close (nsfd);
     }
}


/***********
ARCHIVO: i_tcp_cl.c
Archivo tipo para un cliente con conectores AF_INET, SOCK_STREAM.
***********/

# include “scomun.h”
# include “i_addr.h”

main ( int argc, char *argv [ ]){
     int sfd, nsfd, pid;
     struct sockaddr_in ser_addr;
     nombre_prog =argv[0];
     if((sfd =socket (AF_INET, SOCK_STREAM, 0)) == -1){
          error (DEP, “apertura del conector”);
          exit(-1);
     }
     ser_addr.sin_family =AF_INET;
```

```
     ser_addr.sin_addr.s_addr =
          inet_addr(DIRECCION_NODO_SERVIDOR);
     ser_addr.sin_port =htons (PUERTO_SERVIDOR_TCP);
     if(connect(sfd, &ser_addr, sizeof(ser_addr)) == -1){
          error(DEP, “apertura del conector”);
          exit(-1);
     }
     enviar_mensaje_str(sfd);
     close(sfd);
     exit(0);
}
```

Declaración de funciones para conversión de direcciones.
 # include <sys/socket.h>
 # include <netinet/in.h>
 # include <arpa/inet.h>
unsigned long inet_addr (const char *cp);
Traduce la cadena apuntada por cp, que contiene una dirección en formato decimal, al formato binario.
char *inet_ntoa (struct in_addr in);
Traduce la dirección del formato binario a una cadena con el formato decimal del usuario.

```
/***********
ARCHIVO: i_addr.h
Archivo de cabecera.
***********/

     # ifndef _INTERNETADDR_
          # define _INTERNETADDR_
          # include <stdio.h>
          # include <sys/types.h>
          # include <sys/sockets.h>
          # include <netinet/in.h>
          # include <arpa/inet.h>

          # define PUERTO_SERVIDOR_TCP       7000
          # define PUERTO_SERVIDOR_UDP       7000
          # define DIRECCION_NODO_SERVIDOR   “120.50.3.11”
     # endif
```

```
/***********
ARCHIVO: scomun.h
Archivo de cabecera.
***********/

# ifndef _SCOMUN_
# define _SCOMUN_

# include <stdio.h>
# include <sys/types>
# include <sys/utsname.h>
# include <time.h>
# include <sys/socket.h>
# include <sys/uh.h>
# include <netinet/in.h>

# define CR 10
extern char *nombre_prog;
struct estad{ /* Almacena información del sistema*/
     char nodo [9];
     char sistema [9];
     cahr version [9];
     int pid;
     char fecha [25];
}
```

Asignación de puertos.

Se puede asignar un puerto fijo a un proceso, o bien pedir al sistema que asigne uno que se encuentre libre en ese instante. Asignar un puerto fijo es útil cuando la dirección y el puerto deben ser perfectamente conocidos por los clientes, en este caso se debe tener presente que los números entre el 1 y el 255 están reservados por aplicaciones Internet estándar, los números entre el 256 y el 511 están reservados para aplicaciones futuras de Internet y los números entre el 512 y 1031 están reservados para aplicaciones de usuario ejecutados con privilegios de superusuario. Para utilizar alguno de los puertos se utiliza la llamada rresvport que se define como sigue:

```
     int rresvport (int *port);
```

El puerto reservado es el primero que se encuentre libre en el intervalo [512, 1023].

Los puertos libres para el usuario se encuentran en el intervalo [1024, 65535], es conveniente fijar un número superior al 5000, ya que los otros número son gestionados de forma automática por el sistema, de esta manera aseguramos que no tendremos conflictos con ninguna otra operación.

Manejo del orden de los bytes.

Existe diferencia en la forma como se almacenan los bytes en memoria en algunas máquinas. Existen equipos que utilizan el formato conocido como *big endiand*, en el cual el byte más significativo, se almacena en la posición más baja de la memoria, existe también el formato *little endiand*, el cual es exactamente lo contrario, el byte más significativo, se almacena en la posición menos significativa.

Los protocolos TCP y UDP utilizan el formato big endiand, pero los equipos en la red pueden utilizar cualquiera de la formatos. Para solventar las diferencias en comunicación, se definen las siguientes funciones de conversión de formatos.

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long htonl (unsigned long hostlong);
unsigned short htons (unsigned short hostshort);
unsigned long ntohl (unsigned long netlong);
unsigned short ntohs (unsigned short netshort);
```

`htonl` convierte datos unsigned long del formato del equipo al formato de la red.
`htons` convierte datos unsigned short del formato del equipo al formato de la red.
`ntohl` convierte datos unsigned long del formato de la red al formato del equipo.
`ntohs` convierte datos unsigned short del formato de la red al formato del equipo.

2. RPC

A continuación se presenta un servicio local y el programa que lo invoca. Posteriormente se presenta la conversión a una llamada RPC utilizando la utilería rpcgen de SUN.

```
/* Servicio local. */
#include servicios.h
long servicio_a(long parametro1)
{
long resultado;
resultado = parametro1 + parametro1;
return resultado;
}

/* Archivo de cabecera de servicios. */
/* servicios.h */
long servicio_a(long parametro1, long parametro2);
```

```
/* Programa que invoca el servicio local. */
#include <stdio.h>
#include “servicios.h”

void main (int argc, char *argv[ ] )
{
     long suma;
     long p1;
     if (argc ¡=3) {
          fprintf(stderr, “Uso: %s sumando1 \n”, argv[0]);
          exit(1);
     }

     p1 = (long)atoi(argv[1[);

     suma = servicio_a(p1);
     printf(“Resultado del servicio suma de %f y %f :%f”, p1, p1,
suma);
     exit(0);
}
```

Para la conversión a una llamada RPC es necesario crear un archivo de especificación `servicios.x`.

```
/* servicios.x */
program SERVICIOS_PROG {
     version SERVICIOS_VERS {
          long SERVICIO_A( long, long) = 1;
} = 1;
} = 0x31111111;
```

En el archivo se especifica un número de identificación del servidor remoto 0x31111111, el número de versión del servidor, se inicia típicamente con el número 1 y el número de identificación del servicio, se inicia con el número 1 para el primero.

Para crear los archivos necesarios para el servicio remoto se utiliza el siguiente comando:

```
     rpcgen –C –a servicios.x
```

A partir del comando anterior se generan los siguientes archivos:

```
proto_server.c
proto_svr.c

proto_client.c
proto_clnt.c

proto_xdr.c
proto.h
```

`makefile.proto`

Los archivos `proto_svr.c` y `proto_lnt.c` contienen los talones para el servidor y el cliente, por lo generar no se modifican. El archivo `makefile.proto` se utiliza para compilar el código del cliente y el servidor.

El archivo `proto_client.c` contiene el esqueleto del programa principal del cliente con llamadas ficticias al servicio remoto. Aquí se inserta el código que prepara los argumentos para el servicio remoto.

El archivo `proto_server.c` contiene el talón de los servicios remotos. Aquí se inserta el código para la versión local de los servicios.

A continuación se presenta el archivo `servicios_client.c` generado por el comando `rpcgen` y modificado para incluir la llamada al servicio remoto.

```
#include <stdio.h>
#include “servicios.h”

void main(int argc, char *argv[ ])
{
     long suma;
long p1, p2;
CLIENT * clnt;
if (argc ¡=4) {
          fprintf(stderr, “Uso: %s host sumando1 \n”, argv[0]);
          exit(1);
     }
     clnt = clnt_create(argv[1], SERVICIOS_PROG, SERVICIOS_VERS,
“netpath”);
     if (clnt =0 (CLIENT *)NULL {
          clnt_pcreateerror(argv[1]);
          exit(1);
     }
     p1 = (long)atoi(argv[1]);

     suma = servicio_a_1(&p1, clnt);
     if (suma == (void*) NULL) {
          clnt_perror(clnt, “fallo la llamada”);
     }
     else
          printf(“Resultado del servicio suma de %f y %f :%f”, p1,
p1, suma);
     clnt_destroy(clnt);
exit(0);
}
```

A continuación se presenta el archivo `servicios_server.c` generado por el comando `rpcgen` y modificado para incluir la llamada al servicio remoto.

```
#include “servicios.h”
long * servicio_a_1_svc(long *argp, struct svc_req *rqstp)
{
     static long * result;
     result = *argp + *argp;
     return &result;
}
```

En resumen los pasos a seguir para crear un servicio remoto son los siguientes:

1. Crear el programa utilizando llamados locales.
2. Reestructurar las funciones para que solo tengan un parámetro que se pase por valor.
3. Crear el archivo de especificación con extensión **`.x`**.
4. Generar los archivos necesarios con el comando rpcgen.
5. Modificar los archivos del cliente y el servidor para insertar los llamados y servicios deseados.
6. Compilar los programas con el archivo makefile generado.

3. RMI

A continuación se muestra el código de un servidor de sumas en Java utilizando el mecanismo de invocaciones remotas.

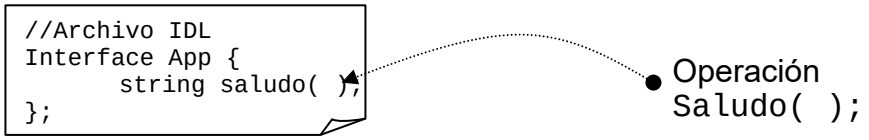
```
// Ejemplo de una invocación a un método en un objeto remoto.

// Interface ServicioSumas
public interface ServicioSumas extends java.rmi.Remote
{
     //Suma de dos números
     public Integer suma2 ( int a, int b)
          throws RemoteExcepion;
     //Suma de tres números
     public Integer suma3 ( int a, int b, int c)
          throws RemoteExcepion;
}

// ServidorServicioSumas
import java.mi.*;
import java.rmi.server.*;
public class ServidorServicioSumas extends UnicastRemoteObject
     implements ServicioSumas
{
     public Servidor ServicioSumas () throw remoteException
     {
          super();
```

```
     }
     //Calcula la suma de dos números
     public integer suma2(int a, int b) throws RemoteException
     {
          s2 = a + b;
          return (s2);
     }
     //Calcula la suma de tres números
     public integer suma3(int a, int b, int c) throws RemoteException
     {
          s3 = a + b + c;
          return (s3);
     }

     public static void main(String args[]) throw Exception
     {
          if (System.getSecurityManager() == null)
               System.getSecurityManager (new RMISecuitManager() );

          //Crea una instancia del servidor de servicio
          ServidorServicioSuma svr = new ServidorServicioSuma
          Naming.bind("ServidorSumas", srv);
          System.out.println ("Servicio encontrado...");
     }
}
```

4. CORBA

La entrada para generar una aplicación CORBA es un archivo de definición de interfaces (archivo.idl). El archivo se coloca en un directorio para la aplicación, el nombre del directorio es el nombre del paquete.

>*directorio_inicial/paquete*/*file*.idl



Se utiliza un compilador para generar código Java ó C++ (stub, skleton)

>idlgen java_poa_genie.tcl –all –jP *paquete App.*idl

Como resultado del comando anterior se obtienen los siguientes archivos:

Cliente:
paquete/client.java

Servidor:
paquete/server.java
paquete/AppImpl.java

En el siguiente ejemplo se muestra la implementación en java en el archivo AppImpl.java

```
//Java
//File 'AppImpl.java'
...
       public java.lang.String saludo()
       throws org.omg.CORBA.SystemException
       {
              java.lang.String _result;

              _result = Hola Liliana!";

              return _result;
       }
}
...
```

Código generado

Implementación del método `saludo( );`

En el archivo client.java se agrega la invocación al método remoto.

```
//Java
//File 'client.java'
...
       try
       {
              ..
              //
              mp_ref = read_referece("App.ref");
              aquete.App App1 = paquete.AppHelper.narrow(tmp_ref);

              System.out.println("El saludo es: " + App1.saludo());

       }
       catch (Exception ex)
       {
              System.out.println("Unexpected CORBA exception: " + ex);
       }
...
```

Código generado

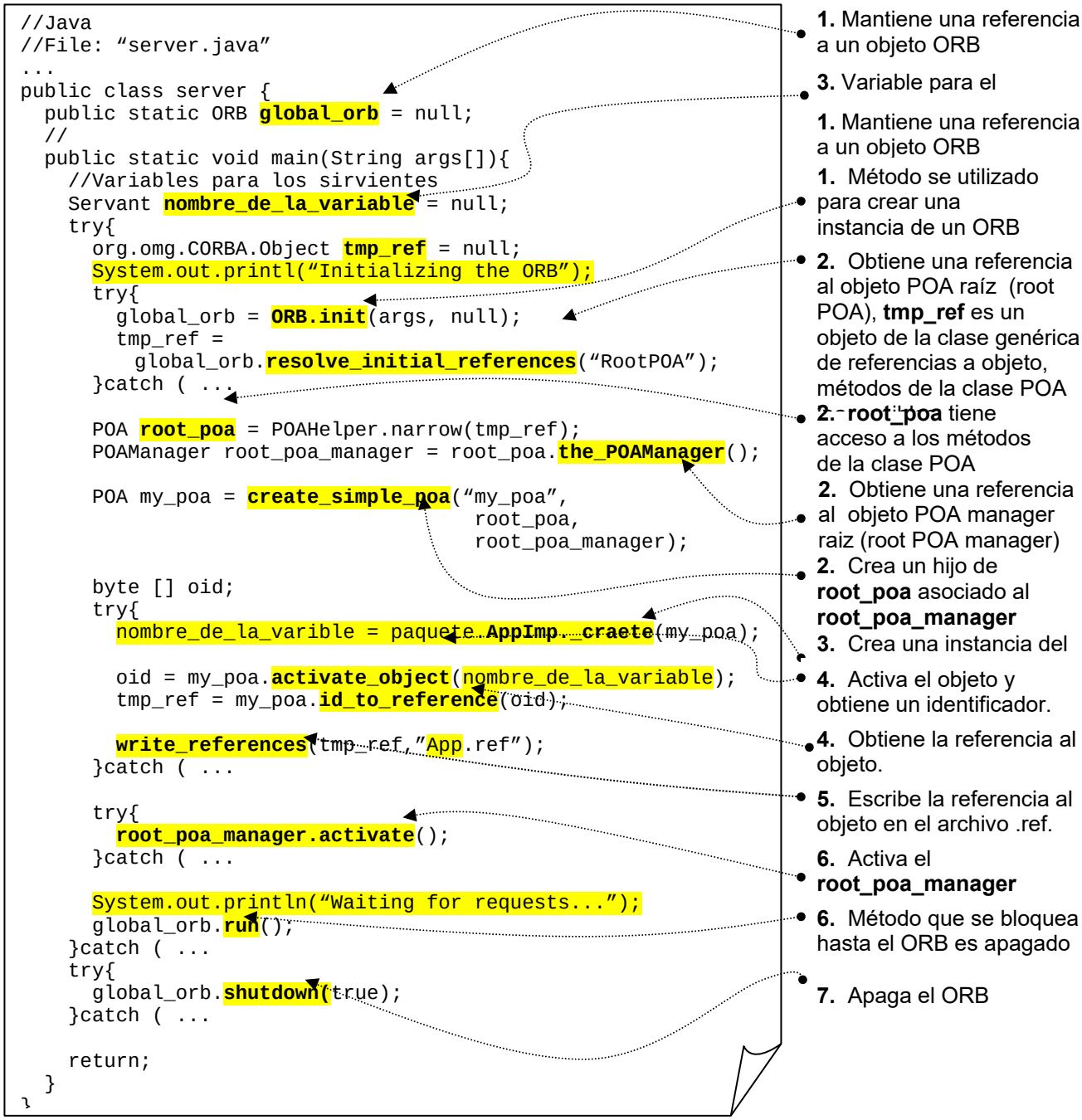
Invocación remota

El archivo **server.java** contiene código para las siguientes tareas:

1. Inicializa el ORB
2. Crea un POA para objeto transeúntes (sólo existen mientras exista el proceso servidor que los creó)
3. Crea los objetos sirvientes.
4. Activa los objetos CORBA, por default un objeto CORBA por cada interfaz definida en el archivo IDL.
5. Exporta las referencias a los objetos, una por cada objeto activo.
6. Activa el ORB, listo para recibir invocaciones.

7. Apaga el ORB

Las notas contienen los números de la tarea a la que apoyan cada uno de los segmentos de código en el archivo.



Dentro del mismo archivo están las funciones invocadas por el main del server( ).

```
//Java
//File: “server.java”
...
public class server {
  public static ORB global_orb = null;

  static void write_reference(org.omg.CORBA.Object ref,
                              String objref_file)
  {
    String strinfied_ref = global_orb.object_to_string(ref);
    System.out.println(
     “Writing strinfied object reference to “ + objref_file);
    try{
      FileWriter store = new FileWriter(objref_file);
      store.write(strinfied_ref);
      store.flush();
      store.close();
    }catch ( ...
  }

  static POA create_simple_poa(String poa_name,
                               POA parent_poa,
                               POAManager poa_manager)
  {
    ...
  }

  public static void main(String args[]){
  ...
  }
}
```

Método sencillo de exportar referencias. El servicio de nombres es ofrece un esquema más

Una vez modificados los archivos se generan las clases con un compilador de java. Hay herramientas, como la herramienta **`ant,`** que toman como entrada los archivos java y genera las clases. En el directorio del paquete se ejecuta el siguiente comando:

```
>ant build_all
```

Una vez generadas las clases, se puede ejecutar la aplicación.

APÉNDICE C: IDL

1. Las interfaces se pueden agrupar en módulos

```
module nombre_del_módulo
{
        interface nombre_de_la interfaz
        {
                // ...
        };
        intarface nombre_de_la_interfaz
        {
                // ...
        };
};
```

2. El nombre completo de las interfaces se forma de la siguiente manera:

   nombre del módulo::nombre de la interfaz

3. Un módulo puede contener módulos, un módulo no puede tener el mismo nombre que el módulo que lo contiene, de la misma forma, una interfaz no puede ser llamada de la misma forma que el módulo que la contiene.

4. Una interfaz contiene operaciones y atributos. Las operaciones describen el comportamiento de un objeto, los atributos son una forma rápida de accesar o seleccionar valores del un objeto.

```
module nombre_del_modulo
{
        interface nombre_de_la_interfaz
        {
                exception nombre_de_la_excepción {};
                //...
                [calificador] atribute tipo identificador;
                [calificador] valor_de_regreso nombre_de_la_operación([dir tipo param1,
...])
                raises (excepción[,excepción,...]);
        };
};
```

5. Una operación define la firma para accesar a los métodos, funciones o procedimientos de un objeto. La firma se compone del valor de regreso, los parámetros (incluyendo tipo y dirección) y las cláusulas de excepción.

6. La dirección *(dir)* de un parámetro puede ser **in** (indicando que es proporcionado por el cliente), **out** (indicando que es proporcionado por el objeto) e **inout** (indicando que es inicializado por el cliente y que puede ser modificado por el servidor)

7. Las operaciones por omisión son síncronas, el cliente que hace la llamada permanece bloqueado hasta obtener la respuesta. Si se agrega la palabra **oneway** el cliente no se bloquea mientras el objeto procesa la llamada. Para usas este tipo de llamados se debe tener un valor de retorno **void**, todos los parámetros debe tener dirección **in** y no utilizar cláusulas de excepción **raises**. No se puede garantizar el éxito de una invocación de este tipo.

8. **oneway void** *nombre_de_la_operación* (**in** *tipo param1*, ...);

9. Un objeto tiene asociada sólo una interfaz, una interfaz puede describir el comportamiento de varios objetos.

10. Los atributos de las interfaces corresponden a las variables definidas en el objeto. Si se agrega la palabra **readonly**, el valor no puede ser modificado por el cliente.

11. Un módulo contiene adicionalmente definición de tipos y definición de constantes.

```
module nombre_del_módulo
{
        typedef tipo_de_dato nombre_del_nuevo_tipo;
        const tipo_de_dato nombre_de_la constane = valor;

        interface nombre_de_la_interfaz
        {
                //...
```

12. Una interfaz puede heredar elementos de otra interfaz base.

```
module nombre_del_módulo
{
        interface nombre_de_la_interfaz_base
        {
                // ...
        };
        interface nombre_de_la interfaz : nombre_de_la_interfaz_base
        {
                // ...
        };
};
```

13. Se pueden utiliza interfaces vacías con el objetivo de agrupar interfaces concretas.

```
module nombre_del_módulo
{
        interface nombre_de_la_interfaz_vacía {};
```

```
        interface nombre_de_la_interfaz : nombre_de_la_interfaz_vacía
        {
                // ...
        };
        intarface nombre_de_la_interfaz : nombre_de_la_interfaz_vacia
        {
                // ...
        };
};
```

14. Todas las interfaces definidas por un usuario heredan elementos de la interface predefinida Object.

15. Una interfaz puede modificar las definiciones heredadas para constantes, tipos y excepciones.

16. Una interfaz debe ser declarada antes de poder ser referenciada por otra interfaz. Se puede hacer una declaración adelantada de interfaces únicamente declarando el nombre de al interfaz.

17. Una interfaz puede ser calificada con la palabra clave **local**. Un objeto que implementa una interfaz local, es un objeto local.

    a. Una interfaz no-local, no puede heredar de un interfaz local.
    b. Los tipos-locales se declaran en interfaces-locales.
    c. Una operación que espera una referencia a un objeto remoto, no puede ser invocado por un objeto local.
    d. No hay ningún tipo de mediación entre invocaciones en un objeto local.

18. Valuetype permite a los programas pasar objetos(referencia) por valor a través de un sistema distribuido.

19. Tipos de datos

Tipo de dato	Tamaño	Rango de valores	Notas
short	>= 16 bits	$-2^{15} \ldots 2^{15} -1$	
unsigned short	>= 16 bits	$0 \ldots 2^{16} -1$	Si la plataforma no soporta enteros nativos de 16 bit, se utiliza el siguiente mas largo tipo de entero soportado
long	>= 32 bits	$-2^{31} \ldots 2^{31} -1$	
unsigned long	>= 32 bits	$0 \ldots 2^{32} -1$	
float	>= 32 bits	IEEE punto flotante	
double	>= 64 bits	IEEE doble precisión	
char	>= 8 bits	ISO Latin-1	0-127 idéntico al código ASCII, 128-255 reservados para caracteres especiales, tales como vocales acentuadas.
string	variable	ISO Latin-1, espera NULO	
string<fija>	variable	ISO Latin-1, espera NULO	Contiene sólo el número de caracteres especificado.
boolean	no especificado	falso o verdadero	
octect	>= 8 bits	0x0 … 0xff	Garantiza que no habrá conversión. Para datos binarios.
any	variable	universal	Tipo definido en tiempo de ejecución.
long long	>= 64 bits	$-2^{63} \ldots 2^{63} -1$	
unsigned long long	>= 64 bits	$0 \ldots 2^{64} -1$	
long double	>= 79 bits	IEEE doble-extendido.	No soportado actualmente por NT.
wchar	no especificado	arbitrario	
wstring	variable	arbitrario	
fixed	no especificado	>= 31 dígitos significativos	
enum	Permite asignar identificadores a los miembros de un conjunto de valores.	CORBA sólo garantiza que los valores ordinales de los valores enumerados se incrementan monótonamente de izquierda a derecha ej. enum Moneda {peso, yen};	
struct	Permite agrupar un conjunto de miembros de diferentes tipo		
union			
array			
sequence			

Tabla C.1. Tipos de datos IDL.

APÉNDICE D: MODELO ISO-OSI

El modelo de referencia OSI es un marco de referencia para la creación de sistemas abiertos con protocolos estándar. El modelo fue desarrollado por la ISO (Organización Internacional de Estándares) y su nombre completo es “International Standards Organization Open Systems Interconnection 7 Layer Reference Model”.

El modelo se divide en siete capas como se muestra en la figura D1, en cada capa se resuelven aspectos particulares de comunicación. El mensaje en la red puede contener encabezados de las diferentes capas.

Cada una de las capas proporciona un servicio a la capa del siguiente nivel y a su vez utiliza los servicios de la capa de menor nivel.

La comunicación en cada una de las capas de lleva a cabo mediante una serie de reglas que conforman un protocolo y que determinan el formato de los datos y el control de la información.

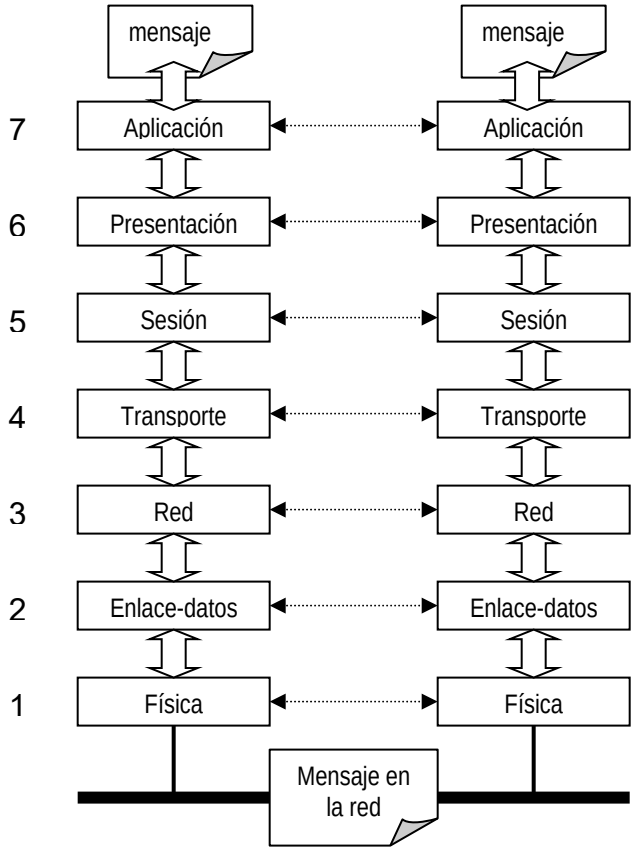


Figura D.1. Capas del Modelo ISO-OSI.

Capa física
Los protocolos en esta capa se concentran en la descripción de los circuitos físico y de la transmisión de bits. Se determinan los voltajes a ser utilizados para representar un 1 ó un 0, el tiempo que debe ocupar un bit, los conectores a utilizar, la velocidad de transmisión

Capa de Enlace de Datos
En esta capa se agrupan los bits y se realizan verificaciones básicas para evitar posibles errores en la transmisión.

Capa de Red
En esta capa se realiza la transferencia de datos entre dos hosts. Los principales aspectos que se tocan en esta capa son el direccionamiento, la elección de rutas y la prevención de cuellos de botella. Los protocolos más utilizados son X25 e IP.

Capa de Transporte
Esta capa esconde todos los detalles de la red a las capas superiores. Se encarga de los mecanismos para el transporte de datos. En esta capa se establece y mantiene una conexión lógica entre. El protocolo de transporte TCP es un protocolo confiable orientado a conexión construido sobre IP. El protocolo UDP es un protocolo sin conexión que sólo hace algunas adiciones menores a IP.

Capa de Sesión
Una sesión es el periodo de tiempo en el cual un usuario permanece lógicamente conectado a un equipo, aún cuando no hay transferencia continua de datos. En esta capa se encuentran protocolos orientados a proporcionar servicios a usuarios finales. Se consideran aspectos como servicios básicos de conexión y estructuras de dialogo entre usuarios.

Capa de Presentación
Los protocolos es esta capa se concentran en el formato de los datos que se están transmitiendo. Se consideran aspectos como traducción y formateo de datos (ej. conversión de ASCII a EBCDIC). En esta capa se pueden incluir aspectos de compresión y encriptación de datos.

Capa de Aplicación
En esta capa se da soporte a aplicaciones distribuidas. Podemos mencionar los protocolos proporcionados para transferencia de archivos y correo electrónico.

En la arquitectura de comunicación de tcp/ip, se utilizan solo las capas física, de red, de transporte y de aplicación.

LISTA DE ACRÓNIMOS

API
Application Program Interface.

COM
Component Object Model. Modelo de Componentes de Microsoft.

CORBA
Common ORB Arquitectura.

DCE
Distributed Computing Environment. Ambiente de Cómputo Distribuido.

DCOM
Distributed COM.

EAI
Enterprise Application Integration. Integración de Aplicaciones Empresariales.

IDL
Interface Definition Language. Lenguaje de Definición de Interfaces.

ISO
International Standard Organization. Organización Internacional de Estándares.

JVM
Java Virtual Machine. Máquina Virtual Java.

MOM
Message Oriented Middleware.

NFS
Network File System. Sistema de Archivos en Red.

OO
Orientado a Objetos.

ORB
Object Request Broker.

OSF
Open System Foundation.

OSI
Open System Interconnect. Interconexión de Sistemas Abiertos.

RMI
Remote Method Invocation. Invocación de Método Remoto.

RPC
Remote Procedure Call. Llamada a Procedimiento Remoto.

SQL
Sequel Query Language.

UML.
Unified Model Language. Lenguaje Unificado de Modelado.

GLOSARIO DE TÉRMINOS

Clase
Prototipo que define variables y métodos que son comunes a los objetos de cierto tipo.

Componente
Pieza de software que implementa un objeto

Comunicación asíncrona
Tipo de comunicación en la cual la aplicación que la inicia se bloquea en espera de una respuesta.

Comunicación síncrona
Tipo de comunicación en la cual la aplicación que la inicia se bloquea en espera de una respuesta.

Evento
Suceso en alguna aplicación que es notificado mediante un mensaje.

Mensaje
Información que viaja de entre dos aplicaciones en comunicación.

Objeto
Pieza de un sistema que consta de variables que contiene su información y métodos que describen su comportamiento.

Proceso
Programa ejecutándose en memoria.

Protocolo
Conjunto de reglas que norman la comunicación entre sistemas.

Proxy
Objeto que hace accesibles de manera local servicios generalmente remotos.

Skeleton
Código que contiene la lógica para tomar peticiones remotas de la red hacer que se ejecuten en el servidor local.

Socket
Puerto de comunicación para la comunicación entre procesos.

Stub
Código que contiene la lógica para las llamadas remotas.

Transacción
Conjunto de operaciones necesarias para completar una tarea.

BIBLIOGRAFÍA

1. **The TUXEDO™ System, Software for Constructing and Managing Distributed Business Applications**; Juan M. Andrade, Mark T. Carges, Terence J. Dwyer, Stephen D. Felts; Addison Wesley; 1996.

2. **CORBA and Software Architecture**; William A. Ruh, Thomas J. Mowbray; EAI Journal May/Jun 1999.

3. **OrbixOTM 3 White Paper**; IONA Technologies, May 1999.

4. **TIB®/Rendezvous™Concepts**, Software Release 6.0, December 1999. (Copyright © 1997-1999 TIBCO Software Inc. ALL RIGHTS RESERVED.)

5. **ENTERPRISE CORBA**; Dirk Slama, Jason Garbis, Perry Russell; Prentice Hall PTR; 1999.

6. **Architecture Support for Global Business Objects: Requirements and Solutions**; Walter Bischofberger, Michael Guttman and Dirk Riehle.

7. **Turbo C/C++ Manual de Referencia**; Hebert Schildt; Mc Graw Hill; 1994.

8. **Tha Java™ Tutorial**; http://javasoft.com/docs/books/tutorial

9. **Common Object Request Broker Architecture (CORBA), v2.4.2**; OMG; February 2001.

10. **Sistemas Operativos Modernos**; Andrew S. Tanenbaum; Prentice Hall Hispanoamericana S.A; 1993.

11. **Practical Smalltalk.**

12. **JAVA 2, Manual de usuario y tutorial**; Agustin Froute; Algaomega RaMa; 2da edición; 2000.

13. **Java, How to Program**; H.M. Deitel, P.J. Deitel; Prenticce Hall; 3th edición.

14. **A UML Patter Languaje**; Paul Evitts; New Riders (ISBN: 157870118X); 2000.

15. **UNIX, programación avanzada**; Fco. Manuel Márquez, AlfaOmega Grupo Editor; 2001.