DISEÑO E IMPLEMENTACIÓN DEL SERVICIO DE EVENTOS DE CORBA PARA JAVA IDL

AUTOR: VÍCTOR M. DÍAZ CABALLERO

2002 **Tutora: Teresa Ariza Gómez**

El objeto de este proyecto fin de carrera es el desarrollo del servicio de eventos distribuidos de CORBA para Java IDL.

En este documento están contenidos la memoria detallada de la realización y funcionamiento de cada una de las partes del proyecto, el código del mismo y el presupuesto de su realización. Se adjunta un CD con el contenido de esta memoria (en formato Word y Acrobat) y todos los elementos necesarios para su instalación y puesta en marcha.

INDICE DE CONTENIDOS:

CAPÍTULO 1:

INTRODUCCIÓN

INTRODUCCIÓN

1. Introducción

En este proyecto se ha implementado uno de los quince servicios distribuidos que se definen en CORBA, en concreto el servicio de eventos. Puesto que CORBA es una especificación y no define ningún detalle relativo a la implementación, deja completa libertad en este aspecto. Ha sido necesario el diseño de toda la estructura de clases y métodos así como la decisión de funcionalidades que CORBA propone como optativas tales como la eliminación de eventos del canal una vez transcurrido un cierto tiempo.

La única limitación ha venido dada por la necesidad de cumplir con las interfaces propuestas por CORBA que permitirán a nuestro servicio interactuar con otras implementaciones tanto del mismo servicio como incluso de otros ORBs.

2. ¿Por qué Java?

La primera decisión a tomar a la hora de implementar un servicio CORBA es la elección del lenguaje a utilizar. En nuestro caso se ha optado por Java que proporciona unas ventajas de sobra conocidas. Java es:

- Simple. Java ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos como pueden ser los punteros, referencias, macros y métodos de liberación de memoria.
- Orientado a objetos. Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las tres características propias del paradigma de la orientación a objetos: encapsulación, herencia y polimorfismo.
- Distribuido. Java se ha construido con extensas capacidades de interconexión TCP/IP. Java en sí no es distribuido, sino que proporciona las librerías y herramientas para que los programas puedan serlo.
- Robusto. Java proporciona comprobación de tipos., declaración explícita de métodos, comprobación de límites de arrays,, excepciones, verificación de byte-codes, etc.
- Arquitectura neutral. Para establecer Java como parte integral de la red, el compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (*run-time*) puede ejecutar ese código objeto
- Seguro. Características como los punteros o el *casting* implícito que hacen los compiladores de C y C++ se eliminan para prevenir el acceso ilegal a la memoria.
- Portable. Más allá de la portabilidad básica por ser de arquitectura independiente, Java construye sus interfaces de usuario a través de un sistema abstracto de ventanas de forma que las ventanas puedan ser implantadas en entornos Unix, Pc o Mac.
- Interpretado. El intérprete Java (sistema run-time) puede ejecutar directamente el código objeto

- Multithreaded. Los threads son básicamente pequeños procesos o piezas independientes de un gran proceso. Al estar los threads construidos en el lenguaje, son más fáciles de usar y más robustos que sus homólogos en C o C++.
- Dinámico. Java no intenta conectar todos los módulos que comprenden una aplicación hasta el tiempo de ejecución. Las librería nuevas o actualizadas no paralizarán las aplicaciones actuales (siempre que mantengan el API anterior).

3. ¿Por qué JavaIDL?

Una vez decidido el lenguaje de programación, una segunda decisión es la del ORB en que se va a basar nuestro servicio. Naturalmente, siempre existe la posibilidad de desarrollar todo el ORB desde un principio, pero esta sería una tarea titánica que no reportaría demasiados beneficios una vez tenida en cuenta la existencia de algunos ya desarrollados por firmas comerciales de gran prestigio. Este es el caso de JavaIDL. JavaIDL es un módulo del lenguaje Java que implementa un ORB CORBA y uno de sus servicios: el de nombrado. JavaIDL es de acceso gratuito por lo que no es necesario realizar ninguna inversión en la compra de licencias.

Las clases Java utilizadas se encuentran en el paquete org.omg.CORBA de la versión Java2 (equivalente al jdk 1.3).

4. Alcance del proyecto

El alcance del proyecto es el diseño e implementación del servicio de eventos; sin embargo, el servicio de eventos sería algo inerte de no existir aplicaciones que lo utilizaran para difundir y/o recibir eventos. Con el propósito de probar el servicio se han incluido en el proyecto aplicaciones que se encargan de utilizarlo. Puesto que, en un principio, puede haber tantas aplicaciones como la imaginación o la necesidad puedan sugerir, y que estas diferirán enormemente unas de otras; las aplicaciones que se han añadido al proyecto se han desarrollado con el único fin de facilitar la realización de pruebas.

Por esto se han desarrollado interfaces de usuario amables utilizando las clases que, a tal efecto, proporciona Java recogidas en su paquete javax.swing.

5. *Estado del arte*

A partir del año 1996, la tecnología CORBA tomó un gran impulso, tanto en la definición de nuevas versiones de la norma como en la aparición de ORBs, ya sean de libre distribución o comerciales. A continuación se citan los más relevantes:

- MICO. La intención de este proyecto es proporcionar una implementación de libre acceso del estándar CORBA bajo C++. Se ha convertido en proyecto tipo OpenSource bastante popular y en la actualidad va por la versión 2.3.7. Implementa entre otros el servicio de nombrado y el de eventos.
- OmniORB es una implementación robusta de un ORB CORBA bajo C++. Es de acceso gratuito y no implementa el servicio de eventos.
- ORBacus está desarrollado por la compañía Object-Oriented Concepts Inc. Permite un *mapeo* para los lenguajes C++ y Java e incluye los servicios básicos de nombrado, eventos y propiedades.
- ORBit está desarrollado en C y no implementa el servicio de eventos.
- TAO es posiblemente el proyecto que más servicios de CORBA implementa entre los que se incluye el de eventos.

La versión 2.0 de CORBA apareció en el mercado en Julio de 1996 y añadía varias características fundamentales:

- Portabilidad de los clientes.
- Extensiones en el interfaz dinámico.
- La posibilidad de operar entre distintos ORBs a través de Internet.
- Extensión de tipos para COBOL.

A partir de este momento se han producido sucesivas mejoras y ampliaciones en la norma hasta llegar a la actual versión 2.2.

CAPÍTULO 2:

LA ARQUITECTURA CORBA

LA ARQUITECTURA CORBA

1. Aspectos generales

El OMG (Object Management Group), define CORBA como la norma que especifica la interoperabilidad entre objetos en un entorno distribuido heterogéneo de manera transparente. De esta definición se obtienen algunas claves para comprender mejor la norma CORBA:

- Al tratarse de una norma pueden existir diversas implementaciones que cumplan los requisitos de compatibilidad con dicha norma, pero que difieran en cuanto a eficiencia, robustez, precio, etc.

- Define mecanismos que hacen posible la interoperabilidad en entornos heterogéneos, considerando a estos en su triple vertiente: lenguaje, plataforma y sistema operativo.

- La transparencia consigue que para el desarrollador de aplicaciones distribuidas sobre plataformas heterogéneas (u homogéneas) todos los aspectos de la red de comunicaciones, los lenguajes de implementación

de los objetos remotos, los sistemas operativos de los *hosts*, etc., queden bajo la responsabilidad de la implementación de CORBA utilizada.

- Por último, es importante destacar que al tratarse de una norma elaborada por una organización plural, en principio independiente, los resultados que se obtienen son siempre fruto del consenso y además accesibles en su totalidad.

La norma CORBA engloba a un conjunto de especificaciones que describen los aspectos que pretende cubrir todo el sistema que se considere compatible CORBA. El alcance de estos se puede resumir en los siguientes puntos:

- Un Lenguaje de Definición de Interfaces: IDL (Interface Definition Language).
- Una infraestructura de distribución de objetos: ORB (Object Request Broker).
- Un conjunto de 15 servicios comunes para los objetos: CorbaServices.[1]
- Un conjunto de servicios comunes para las aplicaciones: CorbaFacilities.
- Un conjunto de servicios especializados en determinados campos: Domain Services
- Una especificación de compatibilidad entre ORBs: IIOP (Internet Inter ORB Protocol).

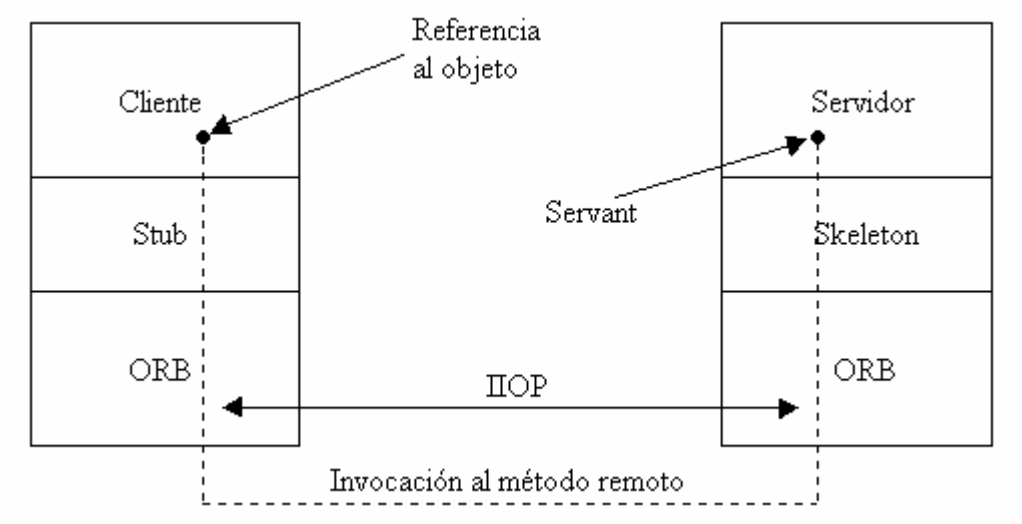
2. La arquitectura CORBA

Cualquier relación entre objetos distribuidos consta de dos extremos: el cliente y el servidor. El servidor proporciona una interfaz remota que es a la que el diente invoca. Hay que destacar que, en este contexto, los términos cliente o servidor se definen a nivel de objeto en lugar de a nivel de aplicación. Una aplicación puede por lo tanto actuar como servidor para algunos objetos y cliente para otros. De hecho un objeto puede ser cliente de la interfaz proporcionada por un objeto remoto

[1] Los distintos servicios son: ciclo de vida, resolución de nombres, gestión de eventos, control de la concurrencia, serialización, relación, transacciones, persistencia, fecha, seguridad, licencia, propiedades, consultas, comercio de objetos, y gestión de configuración.

y, a la vez, implementar una interfaz para ser invocada remotamente por otros objetos.

El siguiente diagrama, muestra una petición realizada desde un cliente CORBA a la implementación de un objeto CORBA en un servidor.



ORB

Para soportar la interacción de objetos entre programas separados CORBA utiliza el ORB (Object Request Broker). El ORB es un objeto que permite la comunicación a bajo nivel entre objetos CORBA. Cualquier objeto que quiera acceder al entorno CORBA necesita darse de alta en un ORB. Una vez establecida esta conexión, el objeto puede utilizar al propio ORB para obtener las referencias de las implementaciones de objetos que estén a su vez conectados.

Cliente

Un cliente es cualquier código (que puede formar parte de un objeto CORBA o no) que invoca un método en un objeto CORBA. El cliente no conoce la totalidad de la estructura del objeto que invoca, tan sólo conoce la parte de ella que corresponde a la interfaz que él posee. Por lo tanto cualquier cambio que se produzca en el objeto remoto no ha de afectar al cliente siempre y cuando no se altere la interfaz.

En el lado del cliente, la aplicación incluye una referencia al objeto remoto. El cliente no tiene acceso directo al ORB sino que accede a él a través del stub que es

un duplicado de los métodos que deben ser invocados remotamente. Cuando el cliente invoca un método en el stub, es éste quién se encarga de invocar las capacidades de conexión del ORB que trasladan la invocación al servidor.

Servant

El servant es una instancia de la implementación del objeto (es decir del código y los datos que implementan el objeto CORBA).

En el lado del servidor, el ORB utiliza el Skeleton para traducir la invocación remota a la llamada al método en el objeto local. El Skeleton traduce la llamada y cualquier parámetro al formato correspondiente en la implementación específica de ese objeto. Cuando el método local termina, el Skeleton transforma el resultado o el error correspondiente y lo envía de vuelta al cliente a través del ORB.

IIOP

La comunicación entre ORBs se lleva a cabo a través de un protocolo compartido denominado IIOP (Internet Inter ORB Protocol), que está basado en el estándar TCP/IP utilizado en Internet. Este protocolo, definido por el OMG, establece cómo las implementaciones de ORBs que cumplan con CORBA deben transmitir la información en uno y otro sentido.

Proceso de invocación

Cuando un ORB recibe una invocación para uno de sus objetos, se realizan las siguientes funciones:

- El ORB inspecciona la referencia del objeto para conocer la identidad de su servant.
- El ORB invoca al Skeleton apropiado, quien dice al ORB cómo decodificar los parámetros para adaptarlos a la implementación concreta del servant.
- El ORB invoca la operación requerida por el cliente en el servant.
- El método en el servant devuelve un resultado al Skeleton.
- El Skeleton codifica el resultado en la forma requerida por el ORB.

- El ORB del servidor devuelve el resultado al ORB del cliente.
- El ORB del cliente devuelve el resultado al stub.
- El stub decodifica el resultado en la forma requerida por el cliente.

3. Interface Definition Language (IDL)

La interfaz IDL define el contrato entre el cliente y el servidor, especificando que operaciones y atributos están disponibles. El lenguaje IDL está definido por el OMG y es independiente del lenguaje de programación en que se vayan a implementar los objetos.

El IDL es un lenguaje exclusivamente declarativo designado para especificar interfaces operacionales para aplicaciones distribuidas. El OMG especifica un mapeo desde IDL hacia diferentes lenguajes tales como C, C++, Smalltalk, COBOL, Ada y por supuesto Java. Al realizar este mapeo, cada declaración de IDL se traduce a la correspondiente declaración en el lenguaje de implementación. Esto posibilita el que se pueda implementar el cliente en un lenguaje y el servidor en otro ya que ambos podrán interoperar a través del ORB como si estuvieran escritos en el mismo lenguaje. Para esto es necesario seguir tres pasos:

- Declarar el módulo CORBA IDL: Un modulo CORBA es un contenedor para interfaces y declaraciones relacionadas. Puede compararse con el Paquete de Java.

- Declarar la interfaz: Al igual que las interfaces en Java, declaran el contrato entre un objeto y el resto con relación a la parte de él que es vista por los demás. Pueden equipararse a las interfaces de Java.

- Declarar las operaciones: Las operaciones CORBA son aquellas que los servidores prometen realizar en nombre de los clientes que lo soliciten. Pueden equipararse a los métodos en Java.

- Declarar las excepciones: Las excepciones CORBA representan una condición excepcional que puede ser devuelta en respuesta a una aplicación. Equivalen a las excepciones Java.

- Declarar los atributos: Un atributo es una variable de acceso público. Al traducir la especificación IDL a Java, el compilador genera un método para la lectura de dicha variable y otro para su escritura (Salvo que se defina como de sólo lectura). Son similares a las propiedades de los JavaBeans.

- Declarar los parámetros: Los parámetros son objetos que se pasan a una operación IDL al ser invocada. Los parámetros pueden ser declarados como "in" (pasan del cliente al servidor), "out" (pasan del servidor al cliente) o "inout" (pasan del cliente al servidor y luego son devueltos por éste al cliente).

CAPÍTULO 3:

JAVAIDL

JAVAIDL

1. *Paso de IDL a Java*

La herramienta idltojava (idlj en la última versión Java2) lee un fichero IDL y crea los correspondientes ficheros Java. Se requiere ejecutar el programa pasando como parámetro el nombre del fichero IDL a compilar y la opción -fAll si se quiere generar el código correspondiente a la parte de servidor y de cliente, es decir si queremos que las clases puedan actuar de ambas formas. Supongamos que se compila el siguiente fichero Hello.idl:

```
module HelloApp
{
   interface Hello
   {
      string sayHello();
   };
};
```

El compilador genera un directorio, con el mismo nombre que el modulo, que contiene 6 ficheros:

Hello.java: Éste es el fichero que contiene la interfaz :

```
package HelloApp;
public interface Hello extends org.omg.CORBA.Object
{
   String sayHello();
}
```

Por supuesto, el objeto Hello debe ser un objeto CORBA para lo cual extiende org.omg.CORBA.Object.

_HelloImplBase.java: Esta clase abstracta es el skeleton del servidor, y le proporciona la funcionalidad básica CORBA. La clase que implemente el servidor debe extender ésta.

_HelloStub.java: Éste es el stub del cliente, y le proporciona la funcionalid básica CORBA.

HelloHelper.java: Esta clase proporciona funcionalidades auxiliares, destacando el método narrow requerido para hacer un "cast" de los objetos CORBA a los tipos correspondientes.

HelloHolder.java: Esta clase final está preparada para contener una instancia pública del tipo Hello. Proporciona la operaciones para utilizarla como argumento out o inout.

HelloOperations.java: En este fichero se declaran los métodos que pueden ser invocados.

2. Clases Holder

Las clases Holder se generan al compilar la especificación IDL, como una ayuda para la implementación de los parámetros out e inout. Necesitamos este recurso ya que Java no soporta este tipo de parámetros que pueden ser modificados en el destino y devueltos al método invocante de la llamada.

La denominación de estas clases se realiza añadiendo el sufijo Holder al nombre del tipo del parámetro. Así la clase Holder para un objeto de tipo Boolean es BooleanHolder. Existen clases ya implementadas para todos los tipos de datos básicos definidos en el paquete org.omg.CORBA.
Cada clase Holder tiene un campo denominado value que contiene la instancia del dato y métodos para leer y modificar este valor.

El funcionamiento es el siguiente: supongamos un método remoto que necesita devolver un String con un determinado nombre y un Boolean con valor true si ese nombre cumple una determinada propiedad. Java sólo nos permite devolver un parámetro por función, así que al invocar el método se pasa como parámetro un BooleanHolder que es rellenado remotamente.

En el cliente se realiza la llamada:

```
String Nombre = new String ();
BooleanHolder boolhold = new BooleanHolder ();
Nombre = serv.rellena_nombre (  boolhold );
```

En el servidor se ejecuta el método

```
Public String rellena_nombre (BooleanHolder boolh )
{
        …
        if ( propiedad )
                boolh.value = true;
        else
                boolh.value = false;
        return Nombre
}
```

En el cliente se tiene el String Nombre con el dato devuelto por el servidor y en el campo value de la variable boolhod el Boolean que indica si cumple la propiedad.

3. Clases Helper

El compilador de IDL genera una clase Helper para cada tipo definido por el usuario. Esta clase tendrá el mismo nombre que el tipo añadiéndole el sufijo "Helper". Por ejemplo y como vimos anteriormente si se define un objeto de tipo Hello, la clase generada es la HelloHelper. Esta clase contiene los métodos

necesarios para manipular instancias de este tipo, en este caso objetos Hello. Estos métodos proporcionan los medios para insertar o extraer el tipo en un objeto tipo Any, obtener el TypeCode del tipo, leer el tipo de un InputStream, escribirlo en un OutputStream, etc.

Sin embargo el método más importante que proporciona esta clase, por ser el más utilizado, es el método narrow.

El método narrow.

Cuando un método devuelve un objeto, lo devuelve como objeto genérico, ya sea extendido de org.omg.CORBA.Object o de java.lang.Object. Antes de poder operar con él, es necesario realizar un "cast" a su tipo real específico. Esto es lo que hace el método narrow, que tiene dos formas, una que opera con un objeto org.omg.CORBA.Object y otra que lo hace con uno java.lang.Object. Cual de los dos métodos proporcione la clase Helper depende de si la interfaz es abstracta o no. Si la interfaz no es abstracta, tendrá un método narrow que acepte un objeto CORBA, mientras que si lo es aceptará un objeto genérico Java.

4. Objetos Any

Un objeto Any sirve como contenedor para cualquier tipo de dato IDL y para cualquier otro que pueda describirse en este lenguaje. Consta de dos partes:

- Un campo value donde se almacena el objeto en sí.
- Un objeto TypeCode que describe el tipo de dato contenido en el campo value.

La instanciación de la clase Any se realiza a través de la clase ORB:

Any data = ORB.init().create_any();

Los objetos Any tienen una gran cantidad de métodos que permiten introducir y extraer cualquier objeto primitivo IDL en él. Por ejemplo, para introducir un String en un Any usamos el método *void insert_string (String s)* que se encarga de copiar

la cadena s al campo value y establecer el valor correspondiente en el objeto TypeCode. De igual forma, para recuperar una instancia de la cadena utilizamos el método *String extract_string ()*; este método lanza la excepción CORBA BAD_OPERATION en caso de que el campo value esté todavía sin establecer o no contenga un objeto String.

Sin embargo, puede que la principal ventaja de un objeto Any es que puede contener cualquier clase que implemente la interfaz *Serializable*. Si consideramos por ejemplo la clase java.awt.Point que implementa dicha interfaz, para introducirla en un objeto Any y posteriormente recuperarla hacemos lo siguiente:

```
// Creamos el punto y el Any
Point point = new Point ( 3, 9 );
Any data = ORB.init().create_any();

// Introducimos el punto
data.insert_Value ( point );

// Modificamos el valor del punto
point.x = 10;
point.y = 15;

//Recuperamos el punto introducido anteriormente
point = data.extract_Value ();

// Ahora el punto vuelve a ser (3, 9)
```

5. *Inicialización*

Para que una aplicación pueda invocar a un objeto CORBA, primero debe crear un ORB e inicializarlo correctamente.

Creación de un ORB para una aplicación

El siguiente fragmento de código muestra como debe crearse un ORB.

```
import org.omg.CORBA.ORB;

public static void main (String args [])
{
        Properties props = new Properties ();
```

```
…
try{
        ORB orb = ORB.init (args, props);
        …
```

Argumentos de ORB.init()

El ORB necesita de cierta información a la hora de inicializarse. La búsqueda de esta información se realiza en tres lugares en el siguiente orden.

- Los parámetros de la línea de comando de la aplicación (args).
- Un objeto de tipo java.util.Properties.
- Un objeto de tipo java.util.Properties devuelto por el método System.getProperties().

El primer valor encontrado por el método init para una propiedad en concreto es el que se utiliza, es decir, si una propiedad no se configura en línea de comando se busca en el objeto Properties que se pase como argumento. Si tampoco está ahí, se toma el valor que devuelva el método System.getProperties().

Actualmente las dos únicas propiedades de configuración definidas para todos los ORBs son:

org.omg.ORBClass: Es el nombre de una clase de Java que implementa la interfaz org.omg.CORBA.ORB. El valor para el ORB de Java IDL es com.sun.CORBA.iiop.ORB y es el que devuelve el método System.getProperties() luego no es necesario especificarlo.

org.omg.CORBA.ORBSingletonClass: Clase utilizada en desarrollo de aplicaciones en entornos seguros. Al igual que la anterior no necesita ser especificada.

Existen otras dos propiedades que soporta Java IDL[1]:

Org.omg.CORBA.ORBInitialHost: El nombre de una máquina donde se ejecutan servidores necesarios para el arranque de aplicaciones como puede ser el servidor de nombres. El valor por defecto para esta propiedad es localhost.

Org.omg.CORBA.ORBInitialPort: El puerto de escucha del servidor de nombres. El valor por defecto es 900.

6. El servicio de resolución de nombres en Java IDL

El Servicio de resolución de nombres proporciona una estructura en árbol para almacenar referencias a objetos CORBA, de forma similar a la estructura que proporciona un sistema de archivos para los directorios. Las referencias a los objetos se almacenan con un nombre. La pareja referencia-nombre se denomina *name binding*. Estos *name binding* se almacenan en *naming context* que son así mismo *name bindings* con la función de organizar la estructura de nombres. El *name context* raíz, del que cuelgan todos los demás, se denomina *initial name context* y es el único que no se elimina cuando el servidor de nombres rearranca.

El servidor de nombres debe estar ejecutándose antes de iniciar las aplicaciones (en caso naturalmente de que utilicen este servicio). Por defecto el puerto de escucha de este servidor es el 900[2].

Adición de un objeto al servidor de nombres

En primer lugar se obtiene el contexto de nombrado inicial:

```
org.omg.CORBA.Object obj =  orb.resolve_initial_references ("NameService");
NamingContext ncRef = NamingContextHelper.narrow(obj);
```

A continuación enlazamos la referencia del objeto en el servidor de nombres:

---

[1] Al especificar una de estas propiedades como argumento en la línea de comandos, es necesario omitir la parte de org.omg.CORBA. Una línea de comando podría quedar -ORBInitialHost 172.126.32.15 -ORBInitialPort 910

[2] Aunque puede cambiarse como se verá posteriormente.

```
NameComponent nc = new NameComponent( Nombre, Contexto );
NameComponent path[] = {nc};
```

Por último hacemos enviamos nuestra referencia para crear el *name binding*:

```
ncRef.rebind( path, this );
```

Hay que destacar que si existiera otra referencia previamente asociada al mismo nombre y contexto que se ha utilizado, esta quedaría sobrescrita. Para evitar esto, hay que cambiar la última línea por:

```
ncRef.bind( path, this );
```

Obtención de una referencia a partir de un nombre

Para obtener una referencia a un objeto, debemos conocer tanto el nombre con que se dio de alta en el servidor de nombres como la localización exacta dentro de su estructura. Supondremos un caso muy simple en el que la referencia se asoció a un nombre en un contexto que cuelga directamente del inicial (caso anterior) .

En primer lugar se obtiene el contexto de nombrado inicial:

```
org.omg.CORBA.Object objRef = orb.resolve_initial_references
("NameService");
```

Se busca la referencia correspondiente al nombre y contexto.

```
NameComponent nc = new NameComponent(Nombre, Contexto);
NameComponent path[] = {nc};
EventChannel eventchannel = EventChannelHelper.narrow
                    (NamingContextHelper.narrow(objRef).resolve(path));
```

Cabe destacar para finalizar, la necesaria utilización del método narrow para convertir los objetos generales CORBA en objetos específicos tal como se especificó anteriormente.

7. *Excepciones CORBA*

CORBA define un grupo de excepciones de sistema estándar, que son lanzadas por el ORB ante errores de muy diverso tipo. Todas las operaciones IDL pueden lanzar excepciones de sistema cuando son invocadas. Al margen de la propia implementación de la operación, el hecho de que la invocación viene de un cliente que corre en otro proceso y en general en otra máquina se traduce en una enorme cantidad de errores posibles. Por lo tanto, un cliente CORBA siempre debe capturar las excepciones de sistema CORBA.

Estructura de una excepción CORBA.

Todas las excepciones de sistema CORBA tienen la misma estructura:

```
exception <SystemExceptionName>
{
        unsigned long minor;
        CompletionStatus completed;
}
```

Las excepciones de sistema CORBA, son subtipos de java.lang.RuntimeException a través de org.omg.CORBA.SystemException.

```
java.lang.Exception
 |
 +--java.lang.RuntimeException  |
    +--org.omg.CORBA.SystemException
       |
       +--BAD_PARAM
       |
       +--//etc.
```

Código Minor

Todas las excepciones de sistema CORBA tienen un campo denominado código minor (minor code), un número que proporciona información adicional acerca de la naturaleza del fallo que ha provocado la excepción. El significado de los diferentes códigos no se especifica por el OMG; cada implementación de ORB especifica los suyos propios.

Estado de conclusión

Todas las excepciones de sistema CORBA tienen un campo denominado estado de conclusión (CompletionStatus) que indica el estado de la operación que lanza la excepción. Estos pueden ser tres:

- COMPLETED_YES: La implementación del objeto ha completado el proceso previamente a lanzar la excepción.
- COMPLETED_NO: Se lanzó la excepción antes incluso de invocar a la implementación del objeto.
- COMPLETED_MAYBE: Se desconoce el estado de la invocación.

CAPÍTULO 4:

EL SERVICIO DE EVENTOS

EL SERVICIO DE EVENTOS

1. Introducción

Una petición CORBA estándar consiste en la ejecución sincronizada de una operación por un objeto. Si la operación define parámetros o devuelve valores, tiene que haber un intercambio de datos entre el cliente y el servidor. Una petición debe ser dirigida a un objeto en concreto y para que tenga éxito, tanto el cliente como el servidor deben estar disponibles. Hay sin embargo algunos escenarios en los que se requiere un modelo de comunicación más elaborado. Por ejemplo:

- Una herramienta de administración de sistemas está interesada en conocer si los discos duros que administra se están quedando sin espacio. El sistema gestor del disco ignora la localización concreta de la herramienta de administración, simplemente informa de que su disco está lleno. Cuando la herramienta de administración recibe el evento, informa al administrador del suceso.

- Una aplicación tiene una lista de propiedades que determinan su funcionamiento. Esta lista está físicamente separada de la aplicación y

puede ser modificada sin necesidad de activar la aplicación. Será necesario pues que cuando la aplicación se active, sea informada de todos los cambios que se han ido produciendo en su lista de propiedades.

- Varios documentos están ligados a una hoja de cálculo. Los documentos están interesados en los cambios que se produzcan en determinadas células de dicha hoja. Cuando estos cambios se produzcan, los documentos actualizarán su presentación basándose en los nuevos valores. De hecho, si un documento permanece inaccesible debido a un fallo durante algún tiempo, cuando se recupere seguirá interesado en conocer todos los cambios producidos en las células de su interés mientras él no estaba activo.

Éstos son sólo algunos ejemplos en los que es interesante la utilización de un servicio de eventos. El servicio de eventos desacopla la comunicación entre objetos definiendo dos roles: productor y consumidor de eventos. Lo productores producen eventos y los consumidores los procesan. Para desacoplar la comunicación entre ambos aparece la figura del canal de eventos.

Un canal de eventos es un objeto intermedio que permite la comunicación asíncrona entre varios productores y varios consumidores. Un canal de eventos es a la vez un productor y un consumidor de eventos, de tal manera que cuando un productor le envía un evento actúa como un consumidor y cuando reenvía este evento a los consumidores actúa como productor. Es obvio decir que los canales de eventos son objetos CORBA estándar y que la comunicación con ellos se realiza utilizando peticiones CORBA estándar.

2. Comunicación de eventos genéricos

Hay dos modelos básicos para comunicar eventos entre productores y consumidores: el modelo Push y el modelo Pull.

Modelo Push

En el modelo Push, los productores envían eventos al consumidor en el momento en que lo consideran oportuno, simplemente invocando la operación push en la interfaz *PushConsumer*.

Para establecer una comunicación de tipo Push, consumidores y productores deben intercambiar referencias de objeto *PushConsumer* y *PushSupplier* respectivamente. Hay que destacar que si bien el consumidor debe proporcionar dicha referencia de forma obligatoria pues el productor necesita conocerla para invocar su método Push, el productor puede ofrecer una referencia nula.

La comunicación puede interrumpirse de dos formas:

- El productor invoca la operación disconnect_push_consumer en la interfaz *PushConsumer*.
- El *consumidor* invoca la operación disconnect_push_supplier en la interfaz *PushSupplier*. Esto sólo será factible si dispone de la referencia al *PushSupplier*, es decir si el productor no ofreció una referencia nula.

La figura 5-1 ilustra una comunicación de tipo Push entre un productor y un consumidor.

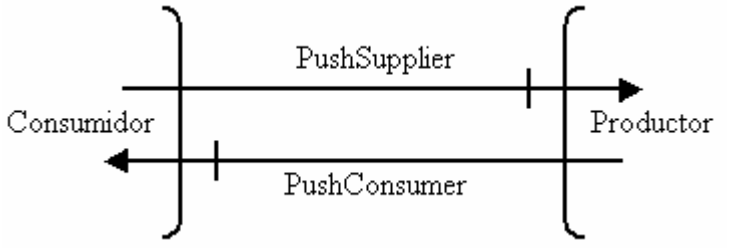


Figura 5-1. Comunicación tipo Push entre un productor y un consumidor.

Modelo Pull

En el modelo Pull, los consumidores piden un evento a los productores; es decir, los consumidores extraen un evento invocando la operación pull en la interfaz *PullSupplier*.

Para establecer una comunicación de tipo Pull, consumidores y productores deben intercambiar referencias de objeto *PullConsumer* y *PullSupplier* respectivamente. Hay que destacar que si bien el productor debe proporcionar dicha referencia de forma obligatoria pues el consumidor necesita conocerla para invocar su método Pull, el consumidor puede ofrecer una referencia nula..

La comunicación puede interrumpirse de dos formas:

- El productor invoca la operación disconnect_pull_consumer en la interfaz *PullConsumer*. Esto sólo será factible si dispone de la referencia al *PullConsumer*, es decir si el consumidor no ofreció una referencia nula.
- El *consumidor* invoca la operación disconnect_pull_supplier en la interfaz *PullSupplier*.

La figura 5-2 ilustra una comunicación de tipo Pull entre un productor y un consumidor.

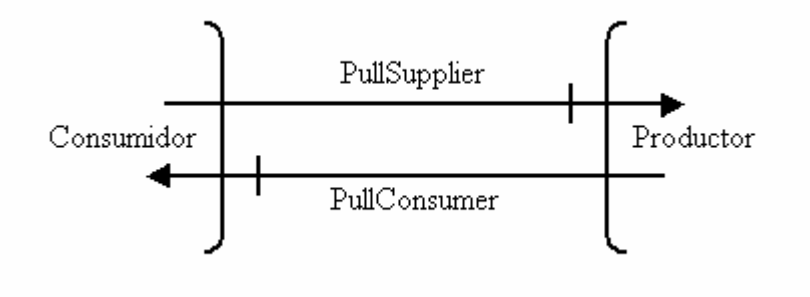


Figura 5-2. Comunicación tipo Pull entre un productor y un consumidor

3. El Módulo CosEventComm

Los tipos de comunicación explicados en el apartado anterior son soportados por cuatro interfaces: *PushConsumer*, *PushSupplier*, *PullSupplier* y *PullConsumer*. Estas interfaces están definidas en un módulo OMG IDL llamado *CosEventComm* que se muestra a continuación.

```
module CosEventComm
{
        exception Disconnected{};

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

        interface PushSupplier
        {
                void disconnect_push_supplier();
        };

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

La interfaz PushConsumer

Un consumidor de tipo Push soporta la interfaz *PushConsumer* para recibir eventos.

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

Un productor comunica un evento al consumidor invocando la operación push y enviando el evento como un parámetro. Si la comunicación ha sido previamente desconectada, se lanza la excepción Disconnected.

La operación disconnect_push_consumer termina con la comunicación y libera los recursos utilizados por el consumidor para soportarla. La referencia al objeto *PushConsumer* es desechada.

La interfaz PushSupplier

Un productor de tipo Push soporta la interfaz *PushSupplier*.

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

La operación disconnect_push_supplier termina con la comunicación y libera los recursos utilizados por el productor para soportarla. La referencia al objeto *PushSupplier* es desechada.

La interfaz PullSupplier

Un productor de tipo Pull soporta la interfaz *PullSupplier* para transmitir eventos.

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

Un consumidor pide un evento al productor invocando bien la operación pull bien la operación try_pull.

- La operación pull bloquea al consumidor hasta que exista un evento disponible o se lance una excepción CORBA. Si la comunicación ha sido previamente desconectada, se lanza la excepción Disconnected.
- La operación try_pull no bloquea al consumidor: si existe un evento disponible, devuelve el evento y establece el valor *true* en el parámetro has_event; si no existe un evento disponible, establece el valor *false* en el parámetro has_event y devuelve un valor indefinido. Si la comunicación ha sido previamente desconectada, se lanza la excepción Disconnected.

La operación disconnect_pull_supplier termina con la comunicación y libera los recursos utilizados por el productor para establecerla. La referencia al objeto *PullSupplier* es desechada.

La interfaz PullConsumer

Un consumidor de tipo Pull soporta la interfaz *PullConsumer*.

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

La operación disconnect_pull_consumer termina con la comunicación y libera los recursos utilizados por el consumidor para establecerla. La referencia al objeto *PullConsumer* es desechada.

4. El canal de eventos (Event Channel)

El canal de eventos es un servicio que desacopla la comunicación entre productores y consumidores. El canal de eventos es a la vez un productor y un consumidor de eventos.

Un canal de eventos puede proporcionar comunicación asíncrona entre productores y consumidores. Aunque consumidores y productores se comunican con el canal de eventos utilizando peticiones CORBA estándar, el canal no tiene porque proporcionar los eventos a los consumidores al mismo tiempo que los recibe de los productores.

Comunicación tipo Push con un canal de eventos

El productor introduce el evento en el canal; el canal a su vez, introduce el evento en el consumidor. La figura 5-3 ilustra una comunicación tipo Push con un canal de eventos.

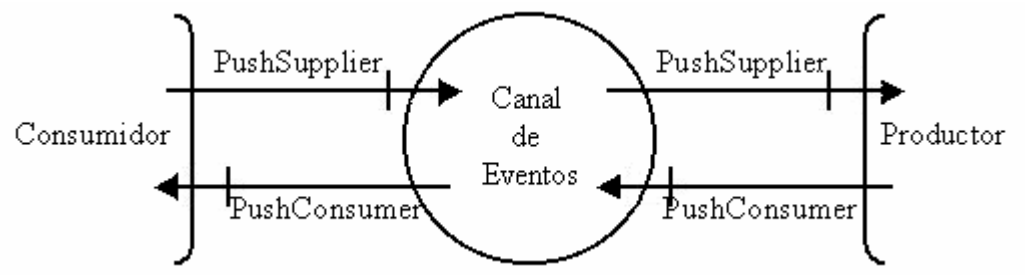


Figura 5-3. Comunicación tipo Push con un canal de eventos.

Comunicación tipo Pull con un canal de eventos

El consumidor extrae eventos del canal; el canal a su vez extrae eventos del productor. La figura 5-4 ilustra una comunicación tipo Pull con un canal de eventos.

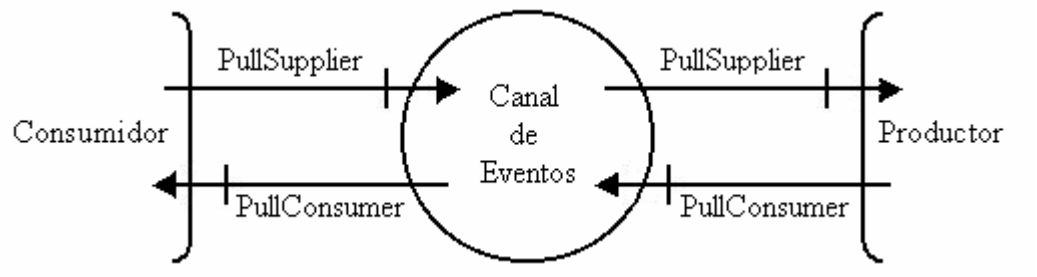


Figura 5-4. Comunicación tipo Pull con un canal de eventos.

Comunicación mixta con un canal de eventos.

Un canal de eventos puede comunicarse con un productor utilizando un tipo de comunicación, y comunicarse con un consumidor utilizando un tipo diferente.

La figura 5-5 ilustra una comunicación tipo Push entre un productor y el canal, y otra tipo Pull entre el canal y un consumidor. El consumidor extrae el evento que el productor ha introducido en el canal.

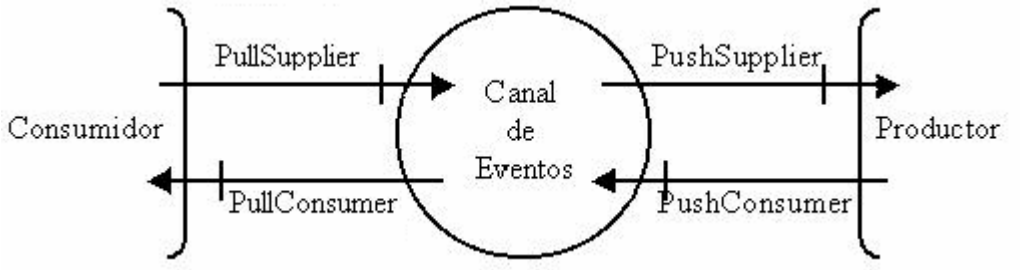


Figura 5-5. Comunicación tipo Push entre un productor y el canal, y tipo Pull entre el canal y un consumidor

Múltiples productores y consumidores

Las figura 5-3, 5-4, y 5-5 ilustran canales de eventos con un solo productor y un solo consumidor. Un canal de eventos también puede proporcionar una comunicación *muchos a muchos*. El canal consume eventos de uno o más productores, y los ofrece a uno o más consumidores.

Un canal de eventos puede soportar la comunicación entre varios consumidores y varios productores sin importar de que tipo sean las comunicaciones de cada uno. Ahora bien, si el canal tiene al menos un consumidor de tipo Push o al menos una petición de tipo Pull, el canal requiere un evento. Por otra parte, si el canal tiene algún productor de tipo Pull, deberá realizarle la demanda correspondiente en espera de que tenga algún evento que ofrecer.

Administración del canal de eventos

El canal de eventos se va construyendo de forma incremental. Recién creado, no existe ningún productor ni consumidor conectado a él. La interfaz *EventChanel* define tres operaciones administrativas: una operación para añadir consumidores, una operación para añadir productores, y una última operación para destruir el canal. Cuando un consumidor quiere conectarse al canal, primero debe conseguir la referencia de un Proxy Supplier. Un Proxy Supplier es similar a un Supplier es decir a un productor (de hecho, hereda su interfaz), pero además incluye métodos para que un consumidor se conecte a él.

Cuando un productor quiere conectarse al canal, primero debe conseguir la referencia de un Proxy Consumer. Un Proxy Consumer es similar a un Consumer es decir a un consumidor (de hecho, hereda su interfaz), pero además incluye métodos para que un productor se conecte a él.

El registro de un productor o un consumidor en un canal es pues un proceso en dos pasos. Un productor primero obtiene del canal la referencia a un Proxy Consumer y después se conecta a él proporcionándole su referencia. De igual forma, un consumidor primero debe obtener la referencia de un Proxy Supplier para después conectarse a él dándole su referencia. El motivo de este método en dos pasos es para poder soportar la conexión de canales de eventos entre sí por parte de un

agente externo. Dicho agente podrá conectar la conexión de dos canales obteniendo un Proxy Supplier de uno y un Proxy Consumer de otro, y pasando a cada uno de ellos la referencia del otro durante la operación de conexión.

Un Proxy puede estar en cualquiera de los siguientes estados: desconectado, conectado o destruido. La figura 5-6 muestra un diagrama con los estados de un Proxy. Los nodos son los estados y las flechas son las operaciones que les hacen pasar de un estado a otro. Por supuesto, las operaciones push, pull y try_pull sólo son válidas en el estado conectado.

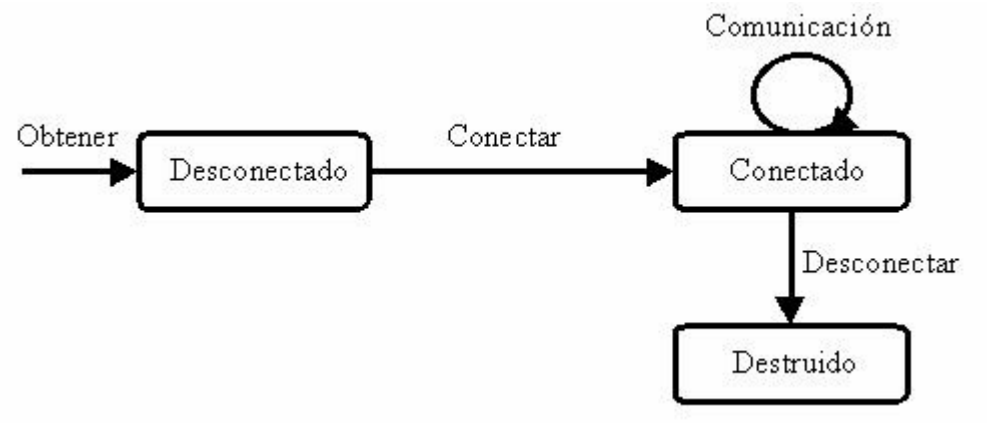


Figura 5-6 Diagrama de estado de un Proxy.

5. *El Módulo CosEventChannelAdmin*

El módulo *CosEventChannelAdmin* define las interfaces para realizar las conexiones entre productores y consumidores. Debido a que alguno de las interfaces (en concreto los de los proxies) descienden de objetos definidos en el módulo *CosEventComm*, es necesario incluir éste.

```
#include "CosEventComm.idl"

module CosEventChannelAdmin
{
        exception AlreadyConnected {};
        exception TypeError {};

        interface ProxyPushConsumer: CosEventComm::PushConsumer
        {
                void connect_push_supplier
```

```
                    (in CosEventComm::PushSupplier push_supplier)
                        raises(AlreadyConnected);
        };

        interface ProxyPullSupplier: CosEventComm::PullSupplier
        {
            void connect_pull_consumer
                    (in CosEventComm::PullConsumer pull_consumer)
                        raises(AlreadyConnected);
        };

        interface ProxyPullConsumer: CosEventComm::PullConsumer
        {
            void connect_pull_supplier
                    (in CosEventComm::PullSupplier pull_supplier)
                        raises(AlreadyConnected,TypeError);
        };

        interface ProxyPushSupplier: CosEventComm::PushSupplier
        {
            void connect_push_consumer
                    (in CosEventComm::PushConsumer push_consumer)
                        raises(AlreadyConnected, TypeError);
        };

        interface ConsumerAdmin
        {
            ProxyPushSupplier obtain_push_supplier();
            ProxyPullSupplier obtain_pull_supplier();
        };

        interface SupplierAdmin
        {
            ProxyPushConsumer obtain_push_consumer();
            ProxyPullConsumer obtain_pull_consumer();
        };

        interface EventChannel
        {
            ConsumerAdmin for_consumers();
            SupplierAdmin for_suppliers();
            void destroy();
        };
    };
```

La interfaz EventChannel

La interfaz *EventChannel* define tres operaciones administrativas: adición de consumidores, adición de productores y destrucción del canal.

```
interface EventChannel
{
        ConsumerAdmin for_consumers();
        SupplierAdmin for_suppliers();
        void destroy();
};
```

Cualquier objeto que posea una referencia a un objeto que soporte la interfaz *EventChannel* puede realizar estas operaciones:

- La interfaz *ConsumerAdmin* permite a los consumidores conectarse al canal de eventos. La operación for_consumers devuelve una referencia a un objeto que soporta la interfaz *ConsumerAdmin*.
- La interfaz *SupplierAdmin* permite a los productores conectarse al canal de eventos. La operación for_suppliers devuelve una referencia a un objeto que soporta la interfaz *SupplierAdmin*.
- La operación destroy destruye el canal de eventos.

El administrador de consumidores y el de productores están definidos como objetos diferentes para que el creador del canal pueda controlar la conexión de productores y de consumidores por separado.

La interfaz ConsumerAdmin

La interfaz *ConsumerAdmin* define el primer paso para la conexión de consumidores al canal. Éstos la utilizan para obtener Proxy Suppliers.

```
interface ConsumerAdmin
{
        ProxyPushSupplier obtain_push_supplier();
        ProxyPullSupplier obtain_pull_supplier();
};
```

La operación obtain_push_supplier devuelve un objeto *ProxyPushSupplier* que es utilizado para conectar un consumidor de tipo Push.

La operación obtain_pull_supplier devuelve un objeto *ProxyPullSupplier* que es utilizado para conectar un consumidor de tipo Pull.

La interfaz SupplierAdmin

La interfaz *SupplierAdmin* define el primer paso para la conexión de productores al canal. Éstos la utilizan para obtener Proxy Consumers.

```
interface SupplierAdmin
{
        ProxyPushConsumer obtain_push_consumer();
        ProxyPullConsumer obtain_pull_consumer();
};
```

La operación obtain_push_consumer devuelve un objeto *ProxyPushConsumer* que es utilizado para conectar un productor de tipo Push.

La operación obtain_pull_consumer devuelve un objeto *ProxyPullConsumer* que es utilizado para conectar un productor de tipo Pull.

La interfaz ProxyPushConsumer

La interfaz *ProxyPushConsumer* define el segundo paso para la conexión de un productor de tipo Push al canal de eventos.

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

Para que un objeto *PushSupplier* se conecte al canal, debe realizar la operación connect_push_suplier. El argumento que debe pasar es su propia referencia aunque, como ya se comentó anteriormente, también puede pasar una referencia nula (en este caso, el canal no podrá invocar la operación disconnect_push_supplier con lo cual el *PushSupplier* será desconectado sin poder ser avisado).

Si el *ProxyPushConsumer* ya ha sido conectado a un *PushSupplier*, se lanza la excepción AlreadyConected.

La interfaz ProxyPullSupplier

La interfaz *ProxyPullSupplier* define el segundo paso para la conexión de un consumidor de tipo Pull al canal de eventos.

```
interface ProxyPullSupplier: CosEventComm::PullSupplier
{
        void connect_pull_consumer
                (in CosEventComm::PullConsumer pull_consumer)
                        raises(AlreadyConnected);
};
```

Para que un objeto *PullConsumer* se conecte al canal, debe realizar la operación connect_pull_consumer. El argumento que debe pasar es su propia referencia aunque, como ya se comentó anteriormente, también puede pasar una referencia nula (en este caso, el canal no podrá invocar la operación disconnect_pull_consumer con lo cual el *PullConsumer* será desconectado sin poder ser avisado).

Si el *ProxyPullSupplier* ya ha sido conectado a un *PullConsumer*, se lanza la excepción AlreadyConected.

La interfaz ProxyPullConsumer

La interfaz *ProxyPullConsumer* define el segundo paso para la conexión de un productor tipo Pull al canal de eventos.

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

Para que un objeto *PullSupplier* se conecte al canal, debe realizar la operación connect_pull_supplier. El argumento que debe pasar es su propia referencia. Si pasa una referencia nula, se lanza la excepción estándar de CORBA BAD_PARAM.

Si el *ProxyPullConsumer* ya ha sido conectado a un *PullSupplier*, se lanza la excepción AlreadyConected.

La interfaz ProxyPushSupplier

La interfaz *ProxyPushSupplier* define el segundo paso para la conexión de un consumidor de tipo Push al canal de eventos.

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

Para que un objeto *PushConsumer* se conecte al canal, debe realizar la operación connect_push_consumer. El argumento que debe pasar es su propia referencia. Si pasa una referencia nula, se lanza la excepción estándar de CORBA BAD_PARAM.

Si el *ProxyPushSupplier* ya ha sido conectado a un *PushConsumer*, se lanza la excepción AlreadyConected.

CAPÍTULO 5:

DISEÑO E IMPLEMENTACIÓN DEL SERVICIO DE EVENTOS DE CORBA

DISEÑO E IMPLEMENTACIÓN DEL SERVICIO DE EVENTOS DE CORBA

1. Introducción

El objetivo de este proyecto es la implementación del servicio de eventos de CORBA para Java IDL. Esta implementación proporciona un canal de eventos con las siguientes características:

- Permite la conexión de productores y consumidores tanto Push como Pull en cualquier combinación y número.

- Permite al creador del canal determinar el tiempo de vida de los eventos, es decir, el tiempo tras el que los eventos serán eliminados del canal.

- Los eventos introducidos en el canal son de objetos de tipo Any que pueden contener, como se detalla en el capítulo tres, cualquier clase que implemente la interfaz Serializable. Esto proporciona una enorme flexibilidad a la hora de enviar datos de cualquier formato.

Llegados a este punto se va a detallar la implementación del servicio de eventos de CORBA, que conlleva tres pasos:

- El primer paso es la compilación del fichero EventService.idl, que genera el código Java de las interfaces que hay que implementar. Esta compilación da lugar a dos paquetes denominados org.omg.CosEventChannelAdmin y org.omg.CosEventComm.

- En segundo lugar se implementan estas interfaces en los paquetes org.omg.CosEventChannelAdminImpl y org.omg.CosEventCommImpl, donde se tiene una clase por cada interfaz a implementar además de algunas clases auxiliares.

- En tercer y último lugar, y para la realización de pruebas, se implementan dos aplicaciones: una dedicada a generar canales y otra que genera clientes para el canal. De esto se encargan los paquetes Applications.EventChannelFactory y Applications.ClientsFactory respectivamente.

2. org.omg.CosEventChannelAdmin.

Este paquete es generado automáticamente por el compilador de IDL, en él se definen las siete interfaces que componen el módulo CosEventChannelAdmin explicado en el capítulo anterior: *EventChannel*, *SupplierAdmin*, *ConsumerAdmin*, *ProxyPushSupplier*, *ProxyPullSupplier*, *ProxyPushConsumer* y *ProxyPullConsumer*. Como ya se vio en el capítulo 3, por cada interfaz se generan seis ficheros.

Además se define una excepción CORBA, AlreadyConnected, para lo cual se generan tres ficheros: AlreadyConnected.java, AlreadyConnectedHelper.java y AlreadyConnectedHolder.java (lógicamente para una excepción no se generan los ficheros correspondientes al stub, Skeleton y operaciones).

3. org.omg.CosEventComm

Este paquete es generado automáticamente por el compilador de IDL, en él se definen las cuatro interfaces que componen el módulo CosEventComm explicado en el capítulo anterior: *PushSupplier*, *PullSupplier*, *PushConsumer* y *PullConsumer*. Como ya se vio en el capítulo 3, por cada interfaz se generan seis ficheros.

Además se define una excepción CORBA, Disconnected para lo cual se generan tres ficheros: Disconnected.java, DisconnectedHelper.java y DisconnectedHolder.java. org.omg.CosEventChannelAdminImpl

4. org.omg.CosEventChannelAdminImpl

Aquí se implementan las interfaces del paquete org.omg.CosEventChannelAdmin. Comprende todas las clases del canal en si: el servant, el administrador de consumidores, el de productores y los cuatro diferentes proxies. Además, contiene las clases auxiliares necesarias para el correcto funcionamiento del sistema.

* EventChannelServant[1]

Esta clase implementa el servant del canal, extendiendo la clase _EventChannelImplBase. Además implementa el interfaz *Runnable*, lo cual quiere decir que lanza un hilo de ejecución propio.

Este objeto es creado por la aplicación EventChannelFactory y permanece activo mientras esta aplicación no lo destruya. Crea los administradores de productores y consumidores y permanece a la escucha de peticiones por parte de los clientes.

```
public EventChannelServant ( EventChannelParams params )
```

[1] En todo el capítulo las clases marcadas con un asterisco son las que implementan las interfaces CORBA. El resto son clases auxiliares.

El constructor recibe como argumento una clase EventChannelParams donde encuentra todos los parámetros que necesita: el nombre del canal, el tipo del canal (nombre del contexto en que se coloca en el servidor de nombres), la referencia del monitor, la dirección del servidor de nombres y su puerto de escucha.

Es aquí donde se crean el administrador de productores y el de consumidores.

* public ConsumerAdmin for_consumers ()[2]

Devuelve la referencia del administrador de consumidores.

* public SupplierAdmin for_suppliers ()

Devuelve la referencia del administrador de productores.

* public void destroy ()

Método que destruye el canal. Llama a las funciones destroy de los administradores para que éstos destruyan a los proxies y estos a su vez a los clientes conectados a ellos. Se desconecta del ORB, elimina la referencia del servidor de nombres y destruye el objeto.

public String Conectar ()

Método encargado de iniciar el ORB y conectar esta clase a él. También se encarga de darle de alta en el servidor de nombres. Si ya existe un objeto con ese nombre sale devolviendo un mensaje explicativo de lo ocurrido, en caso contrario arranca el hilo y devuelve null.

public void run ()

Este hilo no hace nada, tan sólo permanece dormido permitiendo que el servidor permanezca vivo a la escucha de peticiones.

* *SupplierAdminImpl*

Esta clase extiende a _SupplierAdminImplBase implementando al administrador de productores.

Es en esta clase donde se almacenan las referencias de los ProxyPushConsumers y los *ProxyPullConsumers,* esto se hace en dos vectores uno para los proxies tipo Push y otro para los Pull.

public SupplierAdminImpl (EventChannelParams params)

El constructor recibe como argumento una clase EventChannelParams donde encuentra todos los parámetros que necesita: la referencia del monitor y la del administrador de consumidores.

* public synchronized ProxyPushConsumer obtain_push_consumer ()

Método invocado por un PushSupplier como primera fase de su conexión al canal. Crea un ProxyPushConsumer y devuelve su referencia.

* public synchronized ProxyPullConsumer obtain_pull_consumer ()

Método invocado por un PullSupplier como primera fase de su conexión al canal. Crea un ProxyPullConsumer y devuelve su referencia.

public synchronized void ProxyPushDesconecta (ProxyPushConsumerImpl p)

Método invocado por un ProxyPushConsumer para indicar que quiere desconectarse del canal (es decir que el PushSupplier conectado a él quiere desconectarse).

public synchronized void ProxyPullDesconecta (ProxyPullConsumerImpl p)

[2] En todo el capítulo las funciones marcadas con un asterisco son las que definen funciones de las clases que extienden.

Método invocado por un ProxyPullConsumer para indicar que quiere desconectarse del canal (es decir que el PullSupplier conectado a él quiere desconectarse).

public void destroy ( )

Método invocado por el servant cuando recibe la orden de destruirse. Se encarga de eliminar a todos los ProxyConsumers así como de invocar el método disconnect__supplier de estos para eliminar también a los productores.

* *ConsumerAdminImpl*

Esta clase extiende a _ConsumerAdminImplBase implementando al administrador de consumidores.

En esta clase se encuentra la instancia del buffer. Este objeto es de vital importancia ya que es el encargado de almacenar los eventos que recibe el canal.

Es en esta clase donde se almacenan las referencias de los ProxyPushSupplier y los ProxyPullSupplier, esto se hace en dos vectores uno para los proxies tipo Push y otro para los Pull.

public ConsumerAdminImpl (EventChannelParams params)

El constructor recibe como argumento una clase EventChannelParams donde encuentra el parámetro que necesita: la referencia del monitor.

Aquí se crea el buffer.

* public synchronized ProxyPushSupplier obtain_push_supplier ()

Método invocado por un PushConsumer como primera fase de su conexión al canal.

Crea un ProxyPushSupplier y devuelve su referencia.

* public synchronized ProxyPullSupplier obtain_pull_supplier()

Método invocado por un PullConsumer como primera fase de su conexión al canal. Crea un ProxyPullSupplier y devuelve su referencia.

public synchronized void ProxyPushDesconecta (ProxyPushSupplierImpl p)

Método invocado por un ProxyPushSupplier para indicar que quiere desconectarse del canal (es decir que el PushConsumer conectado a él quiere desconectarse).

public synchronized void ProxyPullDesconecta (ProxyPullSupplierImpl p)

Método invocado por un ProxyPullSupplier para indicar que quiere desconectarse del canal (es decir que el PullConsumer conectado a él quiere desconectarse).

public void destroy ( )

Método invocado por el servant cuando recibe la orden de destruirse. Se encarga de eliminar a todos los ProxySuppliers así como de invocar el método disconnect__consumer de estos para eliminar también a los consumidores.

public void elementoEliminadoEnBuffer ()

Método invocado por el buffer cuando elimina un evento que ha quedado obsoleto. Se encarga de incrementar la variable eliminados que es utilizada en el método try_pull.

public void put (org.omg.CORBA.Any dato)

Método que añade el Any que recibe como parámetro al buffer.

public Any try_pull (BooleanHolder nuevoEvento, LongHolder longHold)

Método invocado por un ProxySupplier que devuelve un evento si es que hay alguno disponible o null en caso contrario. Al finalizar, el BooleanHolder contiene true si hay evento y false en caso contrario.

Cuando se invoca el método, el LongHolder contiene el índice del último evento que pidió. Así, el evento que le corresponde al Proxy es el índice menos el número de elementos eliminados (o el primero es caso de que esta resta dé un número negativo).

Al terminar el método el LongHolder contiene el índice actualizado.

* *ProxyPushConsumerImpl*

Esta clase extiende a _ProxyPushConsumerImplBase implementando al ProxyPushConsumer quién se encarga de recibir los eventos del PushSuppliers e introducirlos en el buffer del canal. Además implementa el interfaz *Runnable*, lo cual quiere decir que lanza un hilo de ejecución propio.

public ProxyPushConsumerImpl (EventChannelParams params)

Constructor de la clase. Recibe como argumento una clase EventChannelParams donde encontrará todos los parámetros que necesita: el tamaño de la cola circular que crea, la referencia del monitor, la del administrador de productores y la del administrador de consumidores.

Aquí se crea la cola circular donde se almacenan temporalmente los eventos que se reciben del productor hasta que se introducen en el buffer del sistema.

* public void connect_push_supplier (PushSupplier push_supplier) throws AlreadyConnected

Método utilizado por el PushSupplier para conectarse al canal. Es la segunda y última fase de la conexión. Una vez conectado, se arranca el hilo.

El PushSupplier puede optar por pasar aquí su referencia o pasar un null. De esta decisión dependerá que el canal le informe al destruirse o no.

Se lanza la excepción AlreadyConnected si al llamar al método el PushSupplier ya está conectado al canal.

public void push (Any data) throws Disconnected

Método invocado por el PushSupplier para introducir un evento en la cola circular. Si la cola está llena se bloquea al objeto invocante hasta que quede libre un hueco.

Se lanza la excepción Disconnected si el Push Supplier no está conectado al canal.

public void run ()

Mientras el Proxy permanezca conectado al canal, el hilo saca eventos de la cola y los va introduciendo en el buffer. Para esto, invoca el método get de la cola que lo deja bloqueado mientras no llegue un evento nuevo.

Si se ha recibido la orden de desconexión impide que se sigan introduciendo eventos en la cola aunque antes de terminar saca todos los que ya hubiera y los envía al canal..

public void disconnect_push_consumer()

Método invocado por el PushSupplier (quién sólo transmite una orden del productor) para desconectarse.

Si la cola está vacía introduce en ella un elemento nulo para despertar al hilo que permanece bloqueado en espera de un evento.

public void disconnect_push_supplier ()

Método invocado por el SupplierAdmin para informar al PushSupplier de que lo ha desconectado. Esto será así cuando se destruya el canal. Evidentemente esta

notificación sólo tendrá lugar si el PushSupplier pasó su referencia a este Proxy en la segunda fase.

* *ProxyPushSupplierImpl*

Esta clase extiende a _ProxyPushSupplierImplBase implementando al ProxyPushSupplier que es la parte del canal que ve el Push Consumer y que se encarga de proveerle con los eventos recibidos.. Además implementa el interfaz *Runnable*, lo cual quiere decir que lanza un hilo de ejecución propio.

public ProxyPushSupplierImpl (EventChannelParams params)

Constructor de la clase. Recibe como argumento una clase EventChannelParams donde encontrará todos los parámetros que necesita: la referencia al monitor y al administrador de consumidores.

* public void connect_push_consumer (PushConsumer pco) throws AlreadyConnected

Método utilizado por el PushConsumer para conectarse al canal. Es la segunda y última fase de la conexión. Una vez conectado, se arranca el hilo.

Se lanza la excepción AlreadyConnected si al invocar este método, el PushConsumer ya está conectado al canal.

Se lanza la excepción BAD_PARAM si se recibe null en lugar de la referencia del PushConsumer. Debe conocerse dicha referencia para poder enviarle los eventos.

public void disconnect_push_supplier()

Método invocado por el PushConsumer (quién sólo transmite una orden del consumidor) para desconectarse.

Desconecta al Proxy invocando el método ProxyPushDesconecta del administrador de consumidores

También es invocado por el método run si detecta fallo en la comunicación.

public void disconnect_push_consumer ()

Método invocado por el administrador de consumidores informar al PushConsumer de que lo ha desconectado. Esto será así cuando se destruya el canal.

public void run ()

Mientras el Proxy permanezca conectado, el hilo se encarga de sacar eventos del buffer y enviarlos al PushConsumer. Esto lo hace invocando el método try_pull del administrador de consumidores. Si existe un evento nuevo lo envía al PushConsumer y vuelve a invocar try_pull. Si no existe ningún evento nuevo duerme durante un segundo y lo intenta de nuevo.

* *ProxyPullConsumerImpl*

Esta clase extiende a _ProxyPullConsumerImplBase implementando al ProxyPullConsumer quién se encarga de recibir los eventos del PullSuppliers e introducirlos en el buffer del canal. Además implementa el interfaz *Runnable*, lo cual quiere decir que lanza un hilo de ejecución propio.

public ProxyPullConsumerImpl (EventChannelParams params)

Constructor de la clase. Recibe como argumento una clase EventChannelParams donde encontrará todos los parámetros que necesita: la referencia del monitor, la del administrador de productores y la del administrador de consumidores.

* public void connect_pull_supplier (PullSupplier pull_supplier)

Método utilizado por el PullSupplier para conectarse al canal. Es la segunda y última fase de la conexión. Una vez conectado, se arranca el hilo.

Se lanza la excepción AlreadyConnected si al llamar al método el PullSupplier ya está conectado al canal.

Se lanza la excepción BAD_PARAM si se recibe null en lugar de la referencia del PullSupplier. Debe conocerse dicha referencia para poder pedirle los eventos.

public void run ()

Mientras el Proxy permanezca conectado al canal, el hilo realiza peticiones de tipo Pull al PullSupplier, estos eventos los va introduciendo en el Buffer. Es decir, lo que hace es pedir un evento al PullSupplier, si existe alguno lo envía al Buffer y pide el siguiente. Si no existe ningún evento nuevo permanece bloqueado a la espera.

public void disconnect_pull_consumer()

Método invocado por el PullSupplier (quién sólo transmite una orden del productor) para desconectarse. Su única función es trasladar al administrador de productores esta petición.

public void disconnect_pull_supplier()

Método invocado por el SupplierAdmin para informar al PullSupplier de que lo ha desconectado. Esto será así cuando se destruya el canal. Su única función es informar de este hecho al PullSupplier.

* *ProxyPullSupplierImpl*

Esta clase extiende a _ProxyPullSupplierImplBase implementando al ProxyPullSupplier. Se encarga de esperar peticiones del PullConsumer que pueden ser try_pull o pull. La diferencia radica en que si no hay en el buffer ningún evento nuevo, la petición tipo Pull deja al Proxy bloqueado hasta que llegue alguno mientras que la tipo try_pull no lo bloquea.

public ProxyPullSupplierImpl (EventChannelParams params)

Constructor de la clase. Recibe como argumento una clase EventChannelParams donde encontrará todos los parámetros que necesita: la referencia del monitor y la del administrador de consumidores.

public void connect_pull_consumer (PullConsumer pull_consumer) throws AlreadyConnected

Método utilizado por el PullConsumer para conectarse al canal. Es la segunda y última fase de la conexión.

Se lanza la excepción AlreadyConnected si al llamar al método, el Pull Consumer ya está conectado al canal.

public void disconnect_pull_supplier()

Método invocado por el PullConsumer (quién sólo transmite una orden del consumidor) para desconectarse. Se encarga únicamente de trasladar al administrador de consumidores esta petición.

public void disconnect_pull_consumer ()

Método invocado por el administrador de consumidores para informar al PullConsumer de que lo ha desconectado. Esto será así cuando se destruya el canal. Su única función es informar de este hecho al PullConsumer.

public Any try_pull (BooleanHolder nuevoEvento) throws Disconnected

Método que devuelve un evento si es que hay alguno nuevo disponible. Al terminar, el BooleanHolder contendrá un true si existe dicho evento y un false en caso contrario. Si no hay ningún evento disponible el objeto Any devuelto esta vacío.

La petición del evento se realiza a través del método try_pull del administrador de consumidores.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

public Any pull() throws Disconnected

La invocación de este método deja bloqueado al llamante hasta que esté disponible un evento nuevo que es lo que devuelve.

La petición del evento se realiza a través del método try_pull del administrador de consumidores. Si ésta indica que no hay ningún evento nuevo, dormimos durante un segundo y lo volvemos a intentar.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

Queue

Cola circular de objetos tipo Any. El tamaño de la cola es por defecto de 64 elementos, pero puede establecerse otro al construirla.

public Queue ()

Constructor que no acepta ningún parámetro, genera una cola circular de 64 elementos.

public Queue (int tammax_usu)

Constructor que acepta como parámetro el tamaño de esta cola circular.

public synchronized Any get()

Método que devuelve un elemento de la cola (el siguiente al último que se extrajo). El objeto que invoque este método permanece bloqueado si la cola está vacía, desbloqueándose cuando se reciba un nuevo elemento.

public synchronized void put (Any dato)

Método que introduce un elemento en la cola. Si está llena, el método invocante permanece bloqueado hasta que se produce un hueco.

public boolean llena ()

Método que devuelve un boolean true si la cola está llena y false si no lo está.

public boolean vacia ()

Método que devuelve un boolean true si la cola está vacía y false si no lo está.

Data

Clase auxiliar para el Buffer. Se utiliza para almacenar un objeto Any en el Buffer recordando el momento exacto de su llegada, algo que será vital para determinar en que momento ha de quedar obsoleto.

public Data (Any dato)

Constructor de la clase, realiza una copia del Any que recibe como argumento y almacena la hora GMT de la construcción.

public long GetTime ()

Devuelve la hora GMT en que se construyó este objeto.

public Any GetAny ()

Devuelve el Any que almacena el objeto.

Buffer

Esta clase es el corazón del canal pues es aquí donde se almacenan los eventos recibidos de los productores y de donde se extraen para enviarlos a los consumidores. Además establece un tiempo de vida para los eventos por lo que transcurrido dicho tiempo, el evento es eliminado.

El evento es un objeto de tipo Any que, en lugar de almacenarse tal cual, se introduce en un objeto tipo Data para poder tener constancia de su momento de llegada y por lo tanto poder determinar el momento de su eliminación.

Implementa la interfaz *Runnable*, lo cual quiere decir que lanza un hilo de ejecución propio.

El tamaño de buffer es ilimitado (o al menos limitado sólo por la disponibilidad de memoria).

public Buffer (EventChannelParams params)

Constructor de la clase. Recibe como argumento una clase EventChannelParams donde encontrará todos los parámetros que necesita: la referencia del monitor, la del administrador de consumidores y el tiempo de vida de los eventos en segundos.

Aquí se arranca el hilo.

public synchronized void put (Any dato)

Método que introduce un evento en el Buffer.

public synchronized Any get (int index, BooleanHolder nuevoEvento)

Método que devuelve un evento del buffer, el que ocupa la posición indicada por el parámetro entero. Al terminar, se devuelve el evento (el objeto Any) si es que existe alguno en dicha posición o un Any vacío en caso contrario.

En el BooleanHolder se almacena true si existe el evento o false si no existe.

public void run ()

El hilo se encarga de ver cuando se introdujo el primer elemento del buffer (el más antiguo) y se duerme hasta que llegue la hora de eliminarlo. Una vez eliminado

actúa de igual forma con el nuevo elemento más antiguo. Si el buffer queda vacío, duerme hasta que llegue algún evento.

Una vez eliminado un elemento, avisa al administrador de consumidores a través de su método elementoEliminadoEnBuffer () para que pueda transformar correctamente el índice del Proxy que realice una petición al índice del Buffer.

EventChannelParams

Clase utilizada para almacenar parámetros necesarios para construir el canal de eventos.

Almacena en ella los siguientes parámetros:

- Nombre del canal de eventos.
- Tipo del canal de eventos.
- Dirección de la máquina donde corre el servidor de nombres.
- Puerto de escucha del servidor de nombres.
- Tamaño de las colas existentes en los Proxies.
- Tiempo de vida de los eventos en el canal.
- Referencia del monitor del canal.
- Referencia del administrador de productores.
- Referencia del administrador de consumidores.

```
public EventChannelParams (      String nombre, String tipo, String direccion,
                                 String puerto, int tamaño_colas,
                                 int t_vida_eventos, EventChannelMonitor monitor
                                 )
```

Constructor de la clase. Esta objeto es creado por el objeto que crea al Servant y éste lo recibe como parámetro de construcción. Por esto es necesario establecer desde un principio todos los parámetros excepto las referencias de los administradores (que todavía no se han creado).

```
public EventChannelParams (      String nombre, String tipo, String direccion,
                                 String puerto, int tamaño_colas, int t_vida_eventos
                                 )
```

Constructor idéntico al anterior, la única diferencia es que no al no recibir la referencia de un monitor, construye un monitor básico. Puede ser utilizados por aplicaciones que no necesiten tener control sobre la información de monitorización del canal que por lo tanto no implementen la interfaz EventChannelMonitor.

public void setSupplierAdminRef (SupplierAdminImpl supplier_admin_ref)

Guarda la referencia del administrador de productores, es invocado por el constructor de la clase EventChannelServant.

public SupplierAdminImpl getSupplierAdminRef ()

Devuelve la referencia al SupplierAdmin.

public void setConsumerAdminRef (ConsumerAdminImpl consumer_adminref)

Guarda la referencia del administrador de consumidores, es invocado por el constructor de la clase EventChannelServant.

public ConsumerAdminImpl getConsumerAdminRef ()

Devuelve la referencia al ConsumerAdmin.

public String getNombre ()

Devuelve el nombre que fue almacenado en construcción.

public String getTipo ()

Devuelve el tipo que fue almacenado en construcción.

public String getDireccion ( )

Devuelve la dirección que fue almacenada en construcción.

public String getPuerto ( )

Devuelve el puerto que fue almacenado en construcción.

public int getTamañoColas ( )

Devuelve el tamaño de las colas que fue almacenado en construcción.

public int getTiempoVidaEventos ( )

Devuelve el tiempo de vida de los eventos que fue almacenado en construcción.

public EventChannelMonitor getMonitor ( )

Devuelve la referencia al monitor.

EventChannelMonitor

Interfaz que debe cumplir cualquier clase que se utilice como monitor del canal de eventos.

void Mensaje (int tipo, String Origen, String mensaje)

Este método es invocado por todas las clases del sistema para transmitir al monitor un determinado mensaje, este puede ser de diversos tipos: información general, error, indicación de conexión o de desconexión. Para indicar de que tipo es un mensaje concreto se utilizan las variables finales de la interfaz: INFO, _ERROR, CONEXIÓN y DESCONEXION.

void Accion (int accion)

Método invocado por todas las clases del sistema para informar al monitor de que se ha producido una determinada acción tal como la adición o eliminación de un elemento del buffer, la conexión o desconexión del canal de algún cliente, la destrucción del canal, o el envío o recepción de algún evento. El entero que identifica la acción es una de las siguientes variables finales de la interfaz:

- AÑADIDO_ELEMENTO
- ELIMINADO_ELEMENTO
- CONECTADO_PUSH_SUPPLIER
- CONECTADO_PULL_SUPPLIER
- CONECTADO_PUSH_CONSUMER
- CONECTADO_PULL_CONSUMER
- DESCONECTADO_PUSH_SUPPLIER
- DESCONECTADO_PULL_SUPPLIER
- DESCONECTADO_PUSH_CONSUMER
- DESCONECTADO_PULL_CONSUMER
- CANAL_DESCONECTA
- RECIBIDO_PUSH
- RECIBIDO_PULL
- ENVIADO_PULL
- ENVIADO_PUSH

void insert (Any data)

Una vez que un Consumer (ya sea de tipo Push o Pull) recibe un evento puede pasar, naturalmente,. a utilizarlo de la forma que corresponda a la aplicación concreta. El problema de esto es que hay que modificar su código en cada caso.

Para evitar este inconveniente se ha optado por sacar el evento del propio Consumer y enviarlo al monitor donde será utilizado. Es en éste método dónde se recibe el evento en el monitor.

BasicMonitor

Clase que implementa un monitor muy básico, limitándose a mostrar en la consola los mensajes que recibe. Se ha creado para proporcionar automáticamente un monitor a las aplicaciones que no incorporen el suyo propio.

Hay que destacar que si una aplicación decide no implementar el interfaz EventChannelMonitor deberá modificar el código de los métodos push de la clase PushConsumerImpl y los métodos try_pull y pull de la clase PullConsumerImpl para

utilizar los eventos que reciban ya que, para dotar de mayor generalidad al canal, estas funciones se limitan a pasar los eventos al monitor.

Misc

Clase auxiliar con métodos utilizados en diversos sitios a lo largo del proyecto.
Es una especie de cajón de sastre sin más pretensión que la de no repetir un mismo método en diversas clases.

public void PopUpError (JPanel panel, String Mensaje)

Método que genera un panel de error con el mensaje que se le pase como parámetro.

public JPanel CreaLabelTexto (String label, String descr, JTextField textField, String texto, boolean editable)

Método que crea y devuelve un panel compuesto por un JLabel con el texto label, un JTextField con el texto textField que será editable o no dependiendo del argumento editable.

Los parámetros descr y texto son los mensajes que aparecerán al permanecer el ratón sobre el JLabel y el JTextField respectivamente.

public String TraduceSystemException (org.omg.CORBA.SystemException ex)

Método que devuelve el String explicativo de la excepción de tipo org.omg.CORBA.SystemException que se le pase como parámetro. Este mensaje depende del tipo de la excepción y de su código minor.

5. org.omg.CosEventCommImpl

Aquí se implementan las interfaces del paquete org.omg.EventChannelComm. Comprende los cuatro tipos de productores y consumidores.

Para dotar de mayor generalidad al proyecto los productores no producen ellos mismos los eventos ya que entonces habría que modificarlos para que cada

aplicación enviara su evento concreto. De igual forma, y por el mismo motivo, los consumidores no son el destino final de los eventos sino que estos son enviados al monitor para ser utilizados allí.

* *PushSupplierImpl*

Esta clase extiende a _PushSupplierImplBase implementando al PushSupplier, que se encarga de introducir los eventos que recibe en el canal según el modelo Push. En general, los eventos se podrían producir en esta misma clase pero, para hacerla lo más general posible, se ha optado por generarlos fuera.

public PushSupplierImpl ( )

Un constructor de la clase, no acepta ningún parámetro. El sistema creará un monitor básico que se limitará a mostrar los mensajes en la consola.

public PushSupplierImpl (EventChannelMonitor monitor_ref)

Otro constructor de la clase. Acepta una referencia a un monitor (objeto que implementa la interfaz *EventChannelMonitor*).

public void connect (String Canal, String Tipo, String Direccion, String Puerto, boolean DarRef)

Método invocado por el productor para conectar el PushSupplier al canal de eventos (las dos fases) . Recibe los argumentos:

- Canal: Nombre del canal al que tiene que conectarse.
- Tipo: Tipo del canal al que tiene que conectarse.
- Dirección: Dirección de la máquina donde corre el servidor de nombres.
- Puerto: Puerto de escucha del servidor de nombres.
- DarRef: true si se quiere que el administrador de productores conozca su referencia.

Se lanza la excepción AlreadyConnected si, al invocar el método, el PushSupplier ya está conectado al canal.

public void push (Any dato) throws Disconnected

Este método es invocado por el Productor y se encarga únicamente de introducirlo en el ProxyPushConsumer.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

public void disconnect () throws Disconnected

Método invocado por el Productor cuando quiere desconectar al PushSupplier del canal. Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* public void disconnect_push_supplier ()

Método invocado por el Canal de Eventos para informar de su destrucción. Sólo podrá hacerlo si se le ha dado la referencia de este objeto en la segunda fase de la conexión.

* *PushConsumerImpl*

Esta clase extiende a _PushConsumerImplBase implementando al PushConsumer. Este consumidor una vez conectado al canal recibe todos los eventos que lleguen a él . En general, los eventos se podrían utilizar en esta misma clase pero, para hacerla lo más general posible, se ha optado por generarlos fuera.

public PushConsumerImpl ( )

Un constructor de la clase, no acepta ningún parámetro. El sistema creará un monitor básico que se limitará a mostrar los mensajes en la consola.

public PushConsumerImpl (EventChannelMonitor monitor_ref)

Otro constructor de la clase. Acepta una referencia a un monitor (objeto que implementa la interfaz *EventChannelMonitor*).

public void connect (String Canal, String Tipo, String Direccion, String Puerto) throws AlreadyConnected

Método invocado por el productor para conectar el PushConsumer al canal de eventos (las dos fases). En la segunda fase de la conexión , está obligado a entregar su referencia al Proxy ya que en caso contrario éste no podría enviarle los eventos. Recibe los argumentos:

- Canal: Nombre del canal al que tiene que conectarse.
- Tipo: Tipo del canal al que tiene que conectarse.
- Dirección: Dirección de la máquina donde corre el servidor de nombres.
- Puerto: Puerto de escucha del servidor de nombres.

Se lanza la excepción AlreadyConnected si, al invocar el método, el PushConsumer ya está conectado al canal.

public void disconnect () throws Disconnected

Método invocado por el Consumidor cuando quiere desconectar al PushConsumer del canal.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* public void disconnect_push_consumer ()

Método invocado por el Canal de Eventos para informar de su destrucción. Provoca la destrucción de este objeto.

* public void push(Any data) throws Disconnected

Este método es invocado por el ProxyPushSupplier para introducir un evento en el PushConsumer y de aquí es enviado al consumidor.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* *PullSupplierImpl*

Esta clase extiende a _PullSupplierImplBase implementando al PullSupplier, que se encarga de introducir los eventos que recibe en el canal según el modelo Pull. En general, los eventos se podrían producir en esta misma clase pero, para hacerla lo más general posible, se ha optado por generarlos fuera.

Dispone de una cola circular donde se almacenan temporalmente los eventos que se reciben del productor hasta que los pide el canal.

public PullSupplierImpl ( )

Un constructor de la clase, no acepta ningún parámetro. El sistema creará un monitor básico que se limitará a mostrar los mensajes en la consola.

public PullSupplierImpl (EventChannelMonitor monitor_ref)

Otro constructor de la clase. Acepta una referencia a un monitor (objeto que implementa la interfaz *EventChannelMonitor*).

public void connect (String Canal, String Tipo, String Direccion, String Puerto)
throws AlreadyConnected

Método invocado por el productor para conectar el PullSupplier al canal de eventos (las dos fases). Recibe los argumentos:

- Canal: Nombre del canal al que tiene que conectarse.
- Tipo: Tipo del canal al que tiene que conectarse.
- Dirección: Dirección de la máquina donde corre el servidor de nombres.
- Puerto: Puerto de escucha del servidor de nombres.

En la segunda fase de la conexión debe proporcionar obligatoriamente su referencia al ProxyPullConsumer pues éste la necesita para pedirle los eventos.

Se lanza la excepción AlreadyConnected si, al invocar el método, el PullSupplier ya está conectado al canal.

public void disconnect () throws Disconnected

Método invocado por el productor cuando quiere desconectar al PullSupplier del canal.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* public void disconnect_pull_supplier ()

Método invocado por el Canal de Eventos para informar de su destrucción.

* public Any pull()

La invocación de este método deja bloqueado al llamante hasta que esté disponible un evento (objeto Any) y se le envíe. Este evento se saca de la cola donde lo ha introducido el productor.

* public Any try_pull (BooleanHolder booleanholder)

Método que devuelve un evento si es que hay alguno disponible. Al finalizar el método devuelve el objeto Any y el BooleanHolder contiene true si existe dicho evento y false en caso contrario.

public void push (Any data) throws Disconnected

Con este método, el productor introduce un evento en el PullSupplier, no en el canal. Este evento se almacena en una cola y conforme el canal los demanda se van sacando de ella.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* *PullConsumerImpl*

Esta clase extiende a _PullConsumerImplBase implementando al PullConsumer. Este consumidor no recibe automáticamente los eventos del canal sino que tiene que pedirlos. Una vez recibidos, los eventos se podrían utilizar en esta misma clase, pero para hacerla lo más general posible, se ha optado por utilizarlos fuera.

public PullConsumerImpl( )

Un constructor de la clase, no acepta ningún parámetro. El sistema creará un monitor básico que se limitará a mostrar los mensajes en la consola.

public PullConsumerImpl (EventChannelMonitor monitor_ref)

Otro constructor de la clase. Acepta una referencia a un monitor (objeto que implementa la interfaz *EventChannelMonitor*).

public void connect (String Canal, String Tipo, String Direccion, String Puerto, boolean DarRef)

Método invocado por el productor para conectar el PullConsumer al canal de eventos (las dos fases) . Recibe los argumentos:

- Canal: Nombre del canal al que tiene que conectarse.
- Tipo: Tipo del canal al que tiene que conectarse.
- Dirección: Dirección de la máquina donde corre el servidor de nombres.
- Puerto: Puerto de escucha del servidor de nombres.
- DarRef: true si se quiere que el administrador de consumidores conozca su referencia.

Se lanza la excepción AlreadyConnected si, al invocar el método, el PullConsumer ya está conectado al canal.

public void disconnect () throws Disconnected

Método invocado por el Consumidor cuando quiere desconectar al PullConsumer del canal.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* public void disconnect_pull_consumer ()

Método invocado por el Canal de Eventos para informar de su destrucción. Sólo podrá hacerlo si este objeto proporcionó su referencia al ProxyPullSupplier en la segunda fase de la conexión.

public void pull() throws Disconnected

Este método realiza una petición de tipo Pull al canal de eventos, por lo tanto queda bloqueado por éste hasta que haya un evento disponible. Una vez recibido, el evento se envía al monitor para que sea utilizado allí.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

public void try_pull ( ) throws Disconnected

Este método realiza una petición de tipo try_pull al canal de eventos, por lo tanto no queda bloqueado en ningún momento. En caso de haber algún evento disponible, se envía al monitor para que sea utilizado allí.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

* public void disconnect_pull_consumer ()

Método invocado por el Canal de Eventos para informar de su destrucción. Sólo podrá hacerlo si este objeto proporcionó su referencia al ProxyPullSupplier en la segunda fase de la conexión.

public void pull() throws Disconnected

Este método realiza una petición de tipo Pull al canal de eventos, por lo tanto queda bloqueado por éste hasta que haya un evento disponible. Una vez recibido, el evento se envía al monitor para que sea utilizado allí.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

public void try_pull ( ) throws Disconnected

Este método realiza una petición de tipo try_pull al canal de eventos, por lo tanto no queda bloqueado en ningún momento. En caso de haber algún evento disponible, se envía al monitor para que sea utilizado allí.

Se lanza la excepción Disconnected si no se han completado las dos fases de la conexión antes de invocar este método.

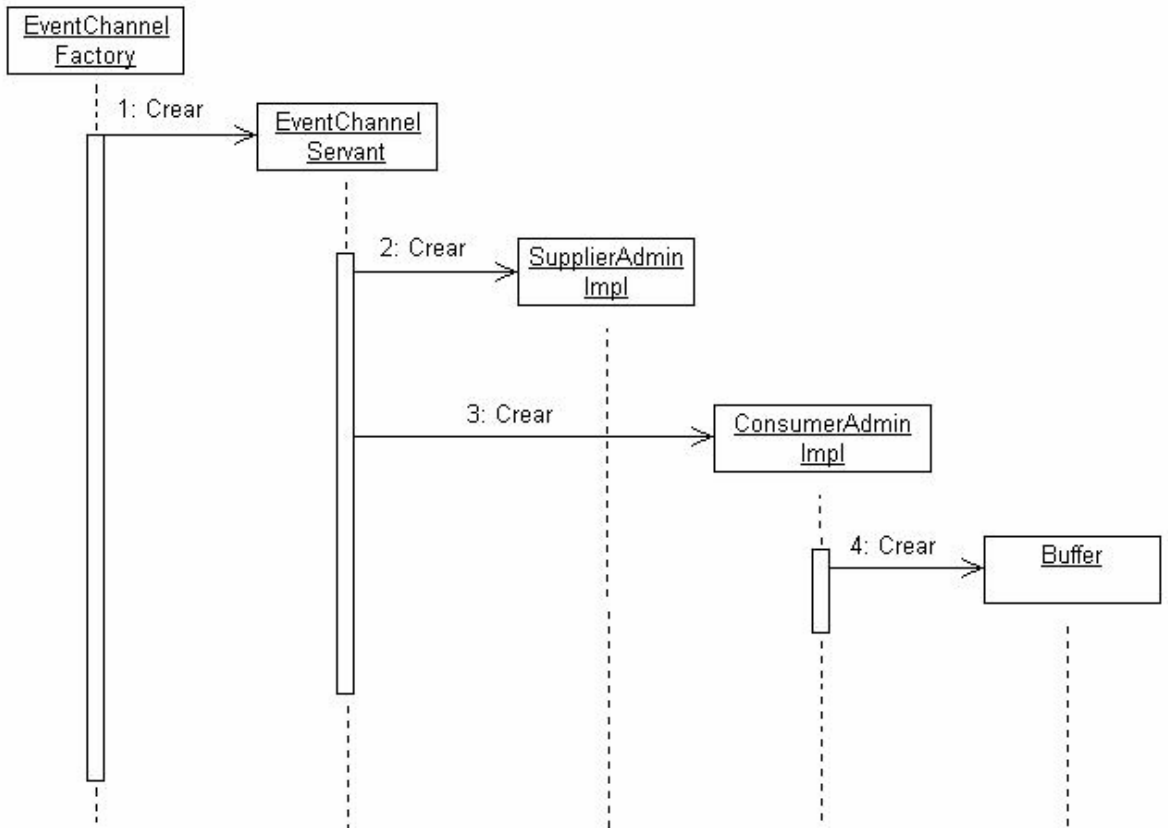
6. Diagramas de secuencia

Los diagramas de secuencia son una parte del sistema de nomenclatura UML (Unified Modeling Language) y muestran interacciones entre objetos según un punto de vista temporal. El contexto de los objetos no se representa de manera explícita, la representación se concentra sobre la expresión de las interacciones.

Un objeto se materializa por un rectángulo y una barra vertical llamada línea de vida de los objetos. Los objetos se comunican intercambiando mensajes representados por medio de flechas horizontales, orientadas del emisor del mensaje hacia el destinatario. El orden de envío de los mensajes viene dado por la posición sobre el eje vertical.

Creación del canal

En este diagrama de secuencia puede verse como a una orden de la aplicación encargada de generar el canal (recordar que el canal no es una aplicación por si mismo), se van construyendo los objetos que son la base del servicio..



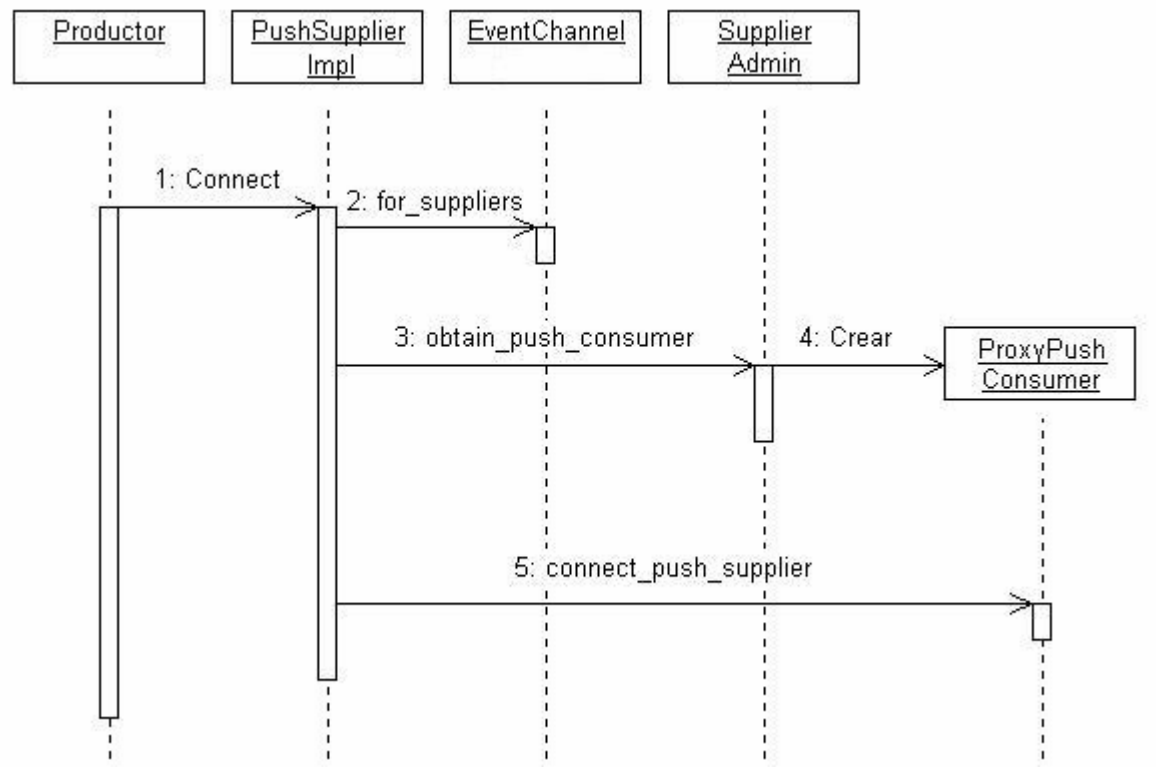
Conexión de un PushSupplier al canal

Una vez que el productor ha creado un PushSupplier, el siguiente paso es que se conecte al canal. Esta conexión se realiza en dos fases. Para poder realizar la primera es necesario que previamente haya obtenido la referencia del canal y, a

través de la invocación del método for_suppliers de éste, la referencia del administrador de productores.

La primera fase de la conexión es la invocación del método obtain_push_consumer del administrador de productores que provoca la creación de un ProxyPushConsumer para el PushSupplier.

Una vez obtenida la referencia del Proxy, el método connect_push_supplier efectúa la segunda fase de la conexión.

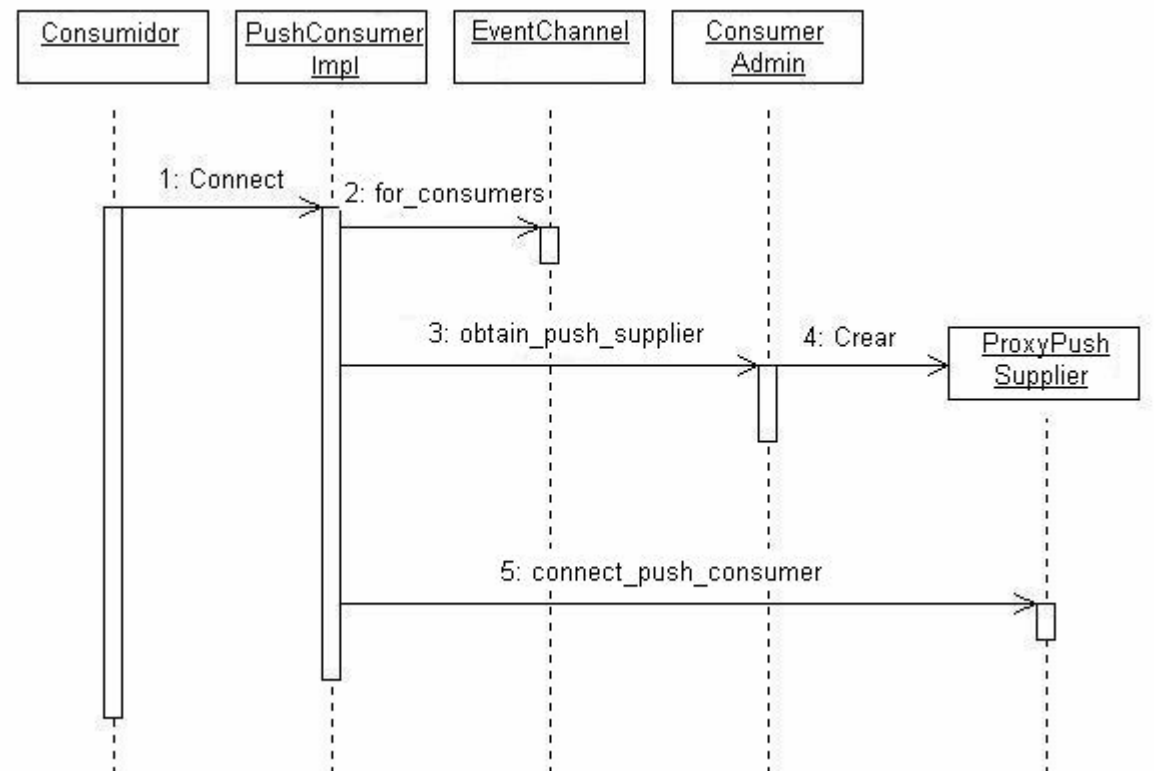


Conexión de un PushConsumer al canal

Una vez que el consumidor ha creado un PushConsumer, el siguiente paso es que se conecte al canal. Esta conexión se realiza en dos fases. Para poder realizar la primera es necesario que previamente haya obtenido la referencia del canal y, a través de la invocación del método for_consumers de éste, la referencia del administrador de consumidores.

La primera fase de la conexión es la invocación del método obtain_push_supplier del administrador de consumidores que provoca la creación de un ProxyPushSupplier para el PushConsumer.

Una vez obtenida la referencia del Proxy, el método connect_push_consumer efectúa la segunda fase de la conexión.

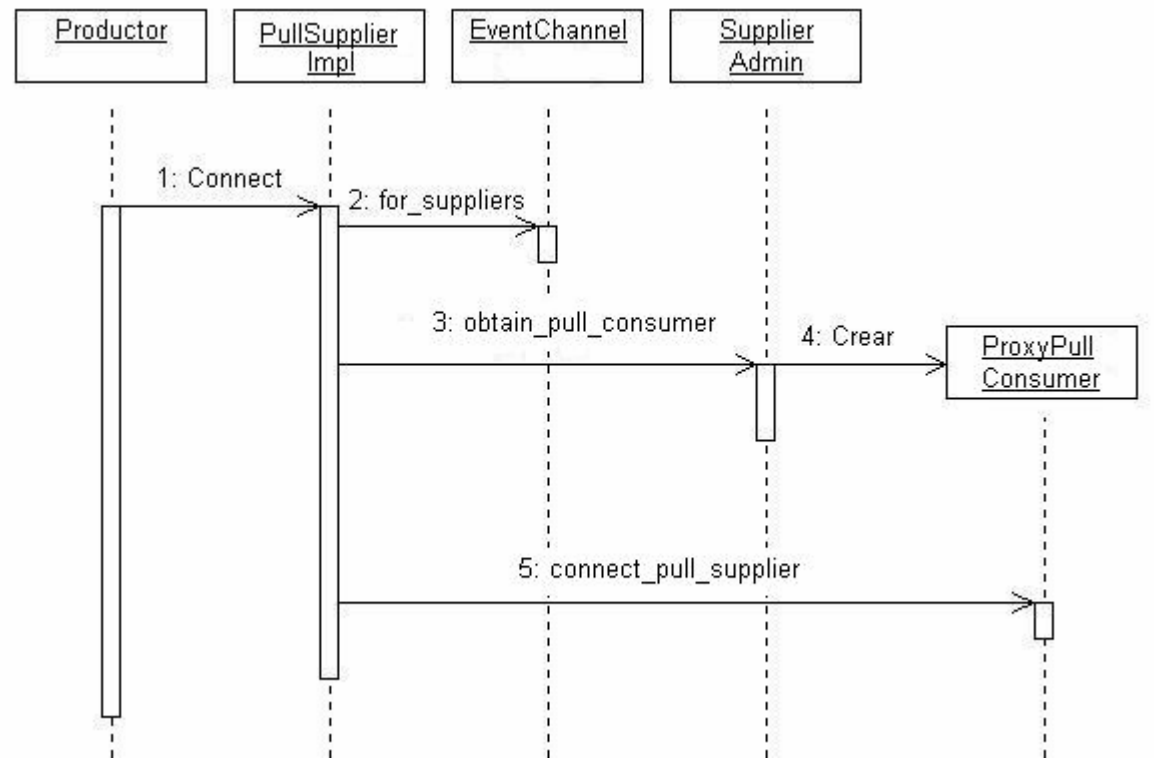


Conexión de un PullSupplier al canal

Una vez que el productor ha creado un PullSupplier, el siguiente paso es que se conecte al canal. Esta conexión se realiza en dos fases. Para poder realizar la primera es necesario que previamente haya obtenido la referencia del canal y, a través de la invocación del método for_suppliers de éste, la referencia del administrador de productores.

La primera fase de la conexión es la invocación del método obtain_pull_consumer del administrador de productores que provoca la creación de un ProxyPullConsumer para el PullSupplier.

Una vez obtenida la referencia del Proxy, el método connect_pull_supplier efectúa la segunda fase de la conexión.

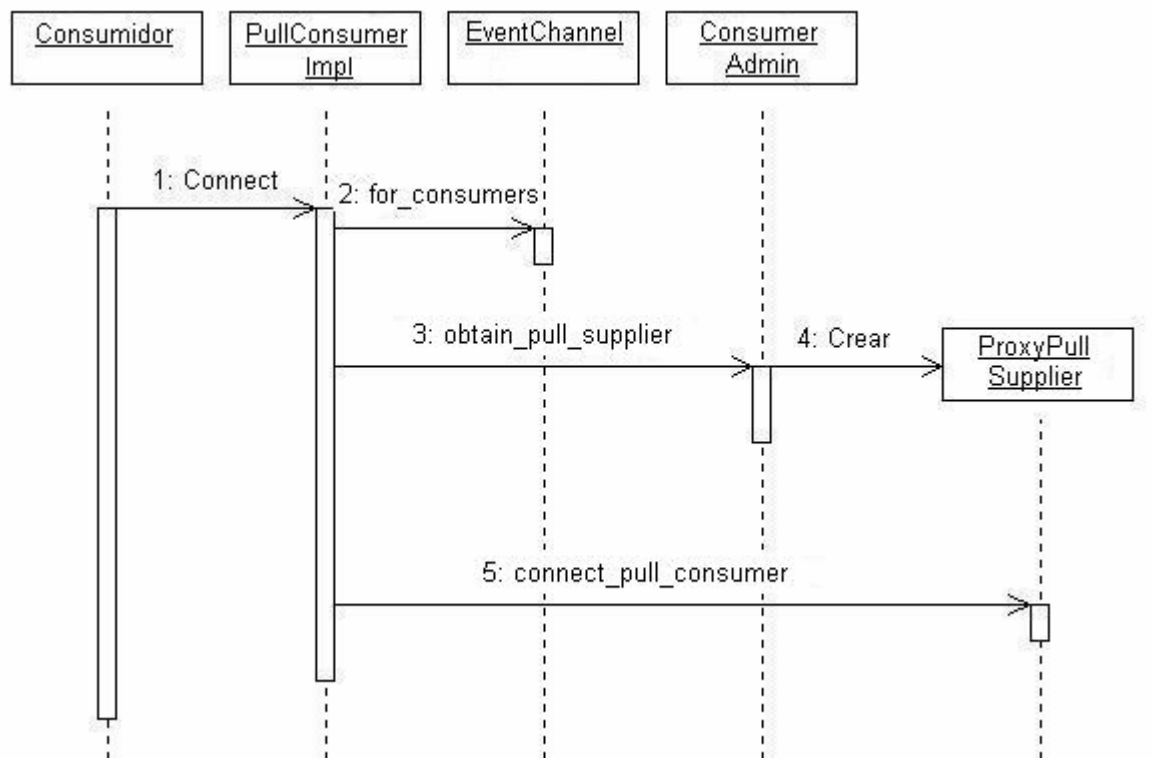


Conexión de un PullConsumer al canal

Una vez que el consumidor ha creado un PullConsumer, el siguiente paso es que se conecte al canal. Esta conexión se realiza en dos fases. Para poder realizar la primera es necesario que previamente haya obtenido la referencia del canal y, a través de la invocación del método for_consumers de éste, la referencia del administrador de consumidores.

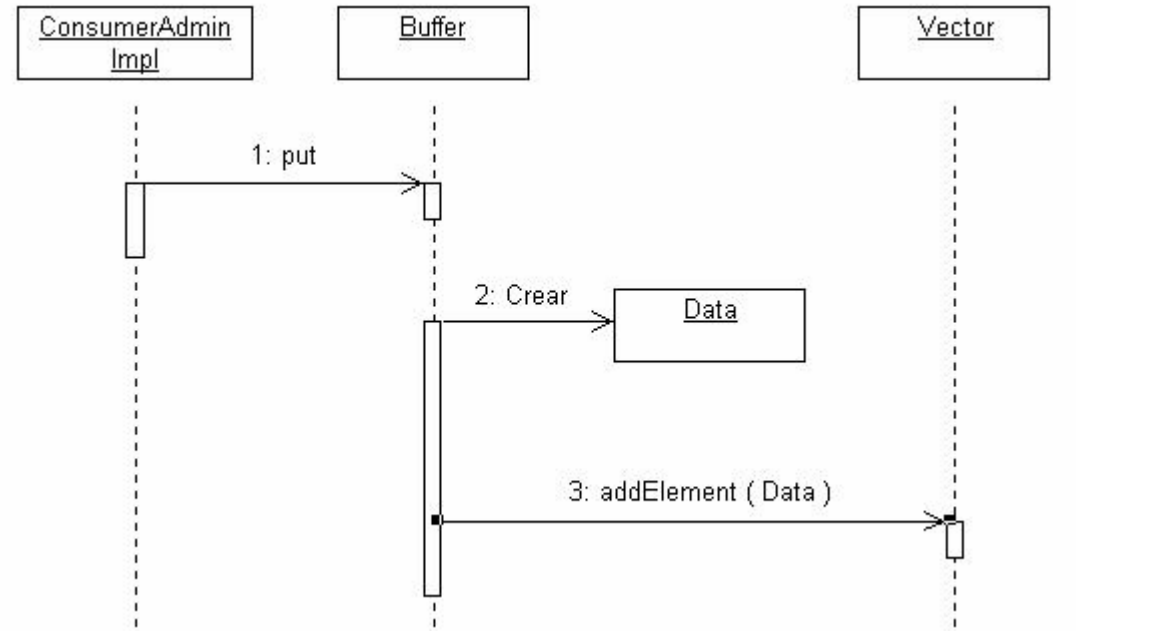
La primera fase de la conexión es la invocación del método obtain_pull_supplier del administrador de consumidores que provoca la creación de un ProxyPullSupplier para el PullConsumer.

Una vez obtenida la referencia del Proxy, el método connect_pull_consumer efectúa la segunda fase de la conexión.



Introducción de un evento en el Buffer

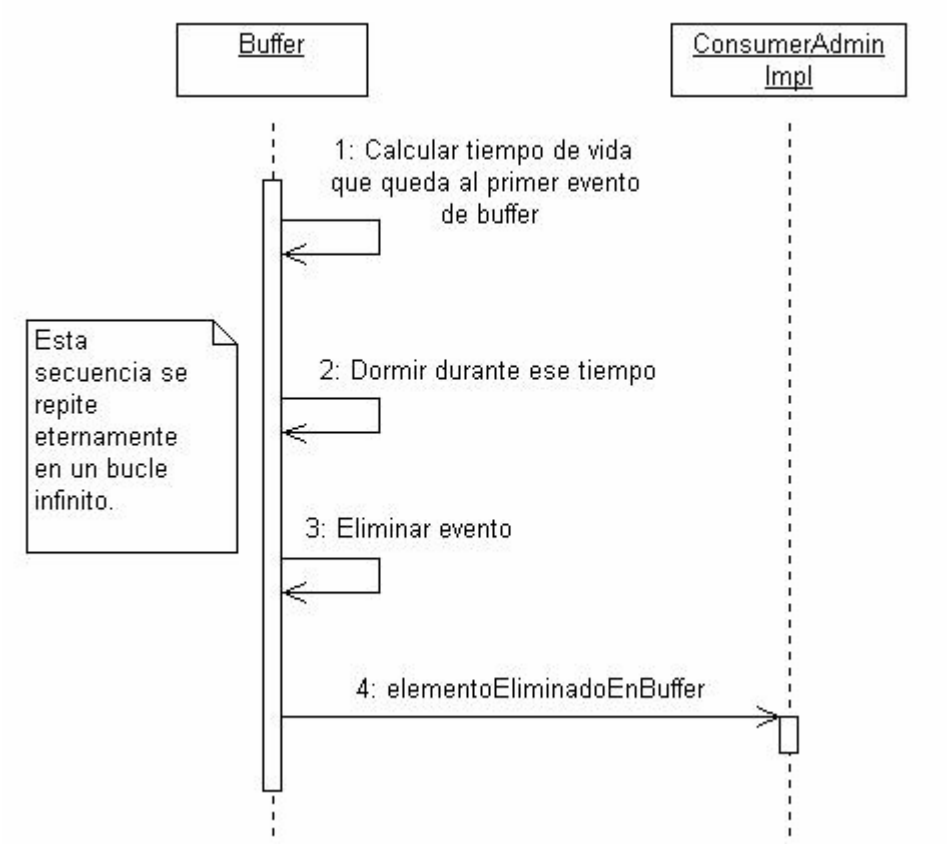
Dado que el Buffer se instancia en el administrador de consumidores, es en este objeto donde se reciben todos los eventos. Una vez recibido uno (un objeto tipo Any), el administrador de consumidores lo introduce en el Buffer con el método put. Una vez en el Buffer se crea un objeto de la clase Data y se añade al vector que actúa como almacén de eventos.



Eliminación de un evento del Buffer

Como ya es conocido, el Buffer tiene un hilo de ejecución propio que se encarga de eliminar los eventos que queden obsoletos. Es necesario recordar que existe un parámetro en la creación de un canal que es el tiempo de vida de los eventos. Cuando entra un evento en el Buffer se registra con él (en el objeto Data que se crea) la hora de su llegada. El hilo del Buffer se limita a ver la hora de llegada del primer elemento y, con el tiempo de vida, calcula cuándo le corresponde eliminarse, durmiendo hasta ese momento. Al despertar, elimina el elemento del Buffer (lo borra del vector) y pasa a realizar el mismo ciclo con el siguiente evento.

Tras eliminar un elemento, se invoca el método elementoEliminadoEnBuffer anteriormente comentado.

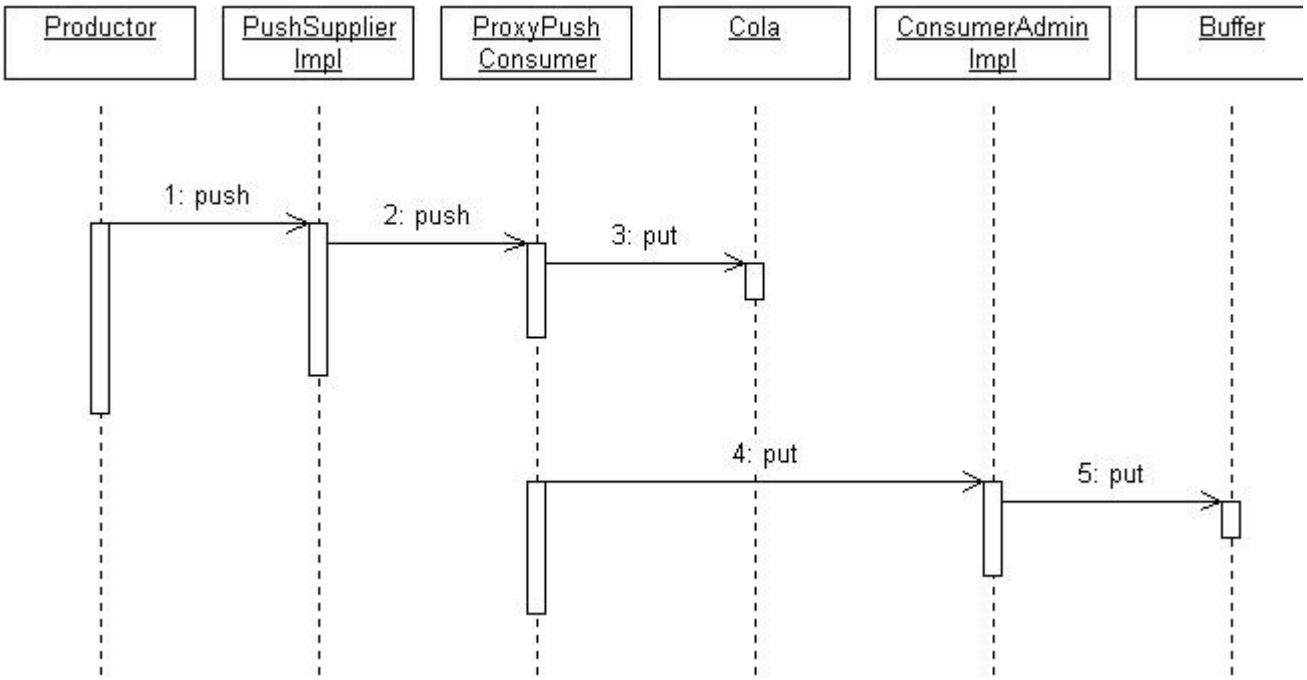


Introducción en el canal de un evento tipo Push

Un productor que tiene asociado un PushSupplier genera un evento y lo introduce en éste con el método push. El PushSupplier introduce el evento en su Proxy

asociado invocando también su método push. El Proxy introduce el evento en una cola circular para poder liberar al productor en caso de que el canal esté ocupado realizando otras gestiones.

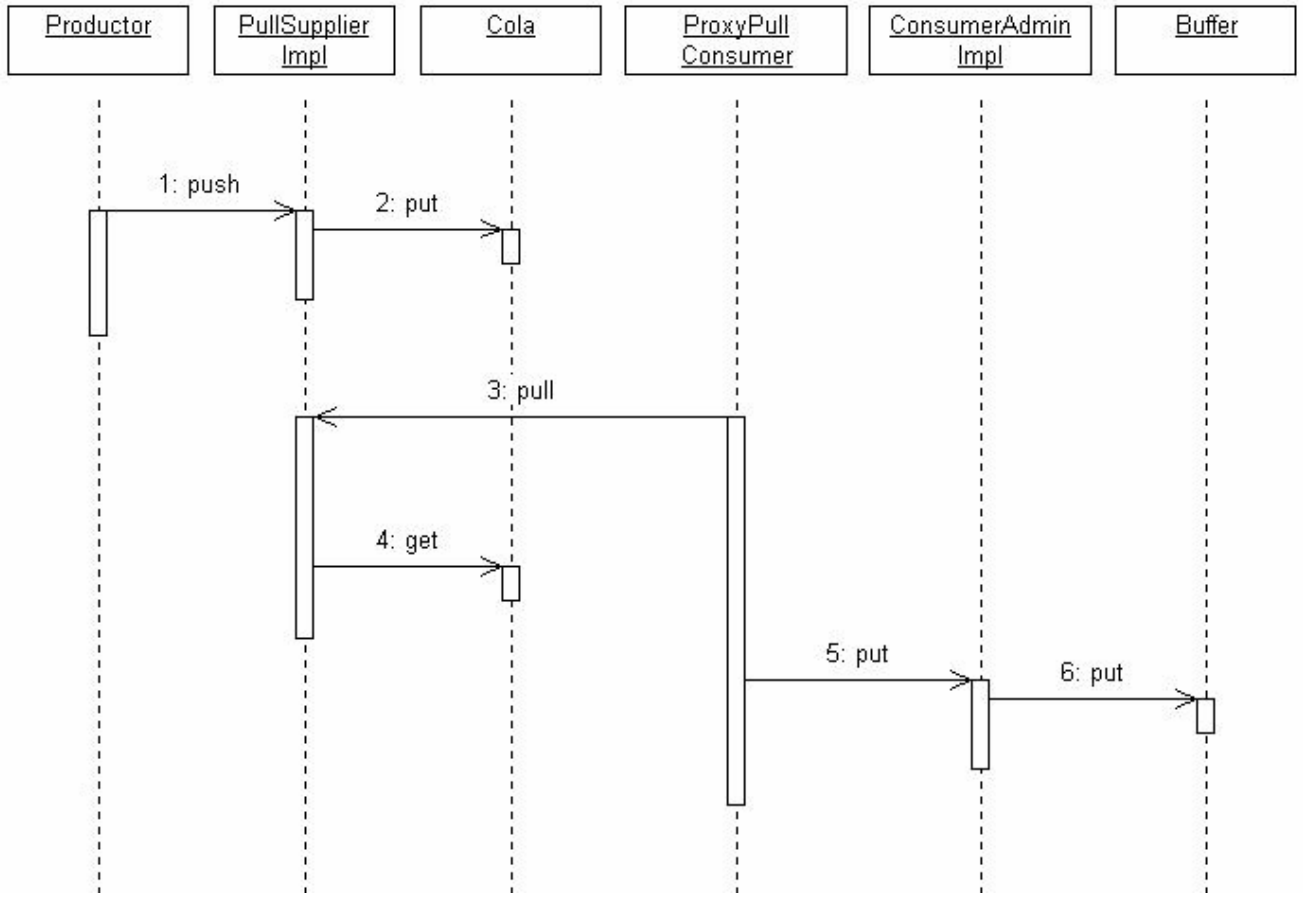
El hilo de ejecución propio que posee cada ProxyPushConsumer se dedica a extraer los eventos de la cola circular e introducirlos en el administrador de consumidores quién a su vez lo introduce en el Buffer como ya se ha explicado.



Introducción en el canal de un evento tipo Pull

Un productor que tiene asociado un PullSupplier genera un evento y lo introduce en éste con el método pull. El evento es introducido en una cola circular puesto que es el canal quién debe pedirlo.

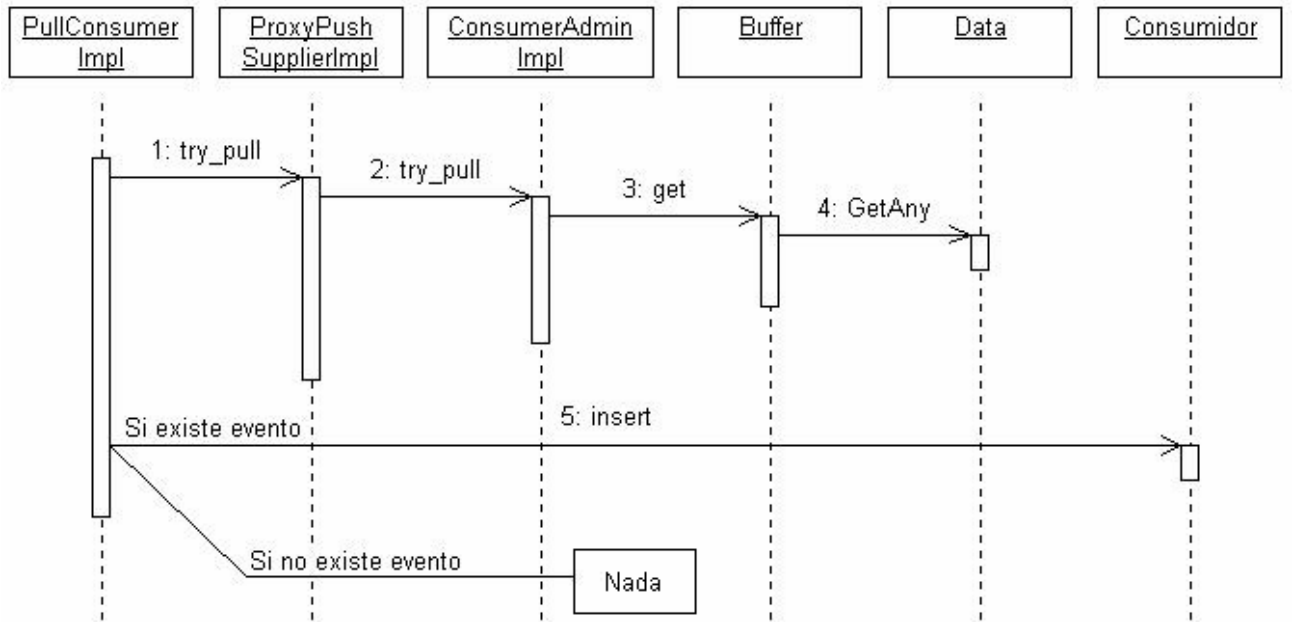
Tras haber pedido el evento, el ProxyPullConsumer lo recibe y lo introduce en el administrador de consumidores con el método put. El resto es igual que en el caso anterior.



Extracción del canal de un evento por el método TryPull

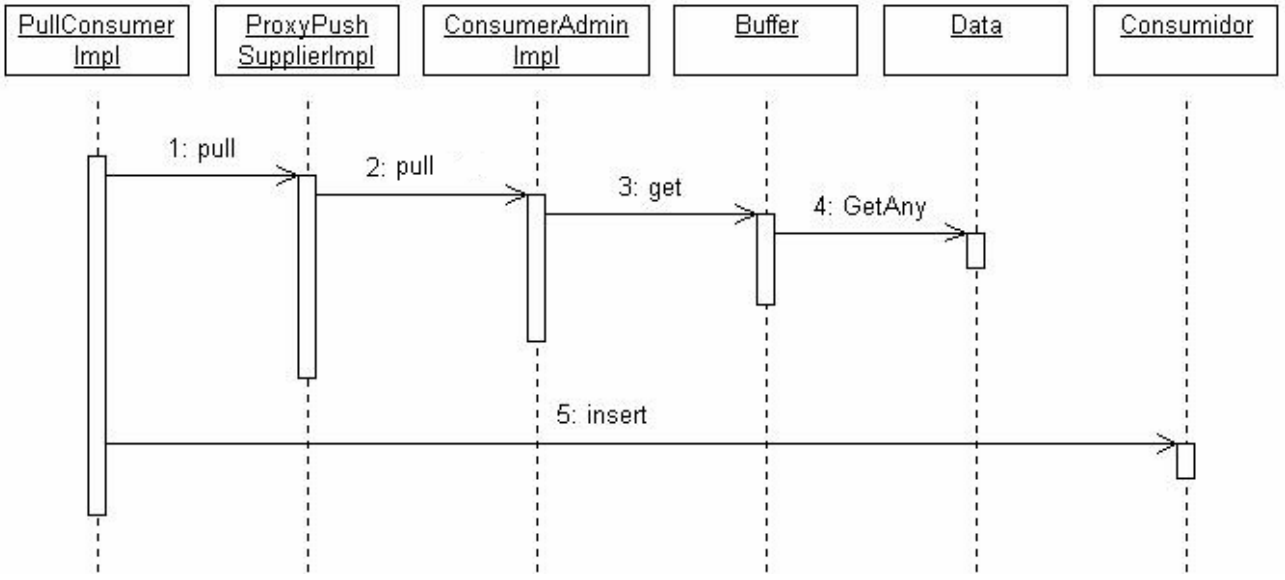
La extracción de un evento del canal por el método TryPull, conlleva que deba ser el PullConsumer quién lo demande. Una vez realizada esta petición a su ProxyPullSupplier a través del método try_pull, el Proxy la traslada al administrador de consumidores invocando el método con el mismo nombre que éste posee. El administrador de consumidores realiza a su vez la petición al Buffer para lo cual tiene en cuanta el índice de eventos demandados por el Proxy y los eventos eliminados del Buffer como se explica anteriormente.

El Proxy devuelve siempre un evento al PullConsumer pero puede estar vacío (recordar el uso del BooleanHolder). Por lo que sólo en el caso de que el Booleanholder indique que se trata realmente de un evento nuevo el PullConsumer lo introduce en el consumidor invocando su método insert.



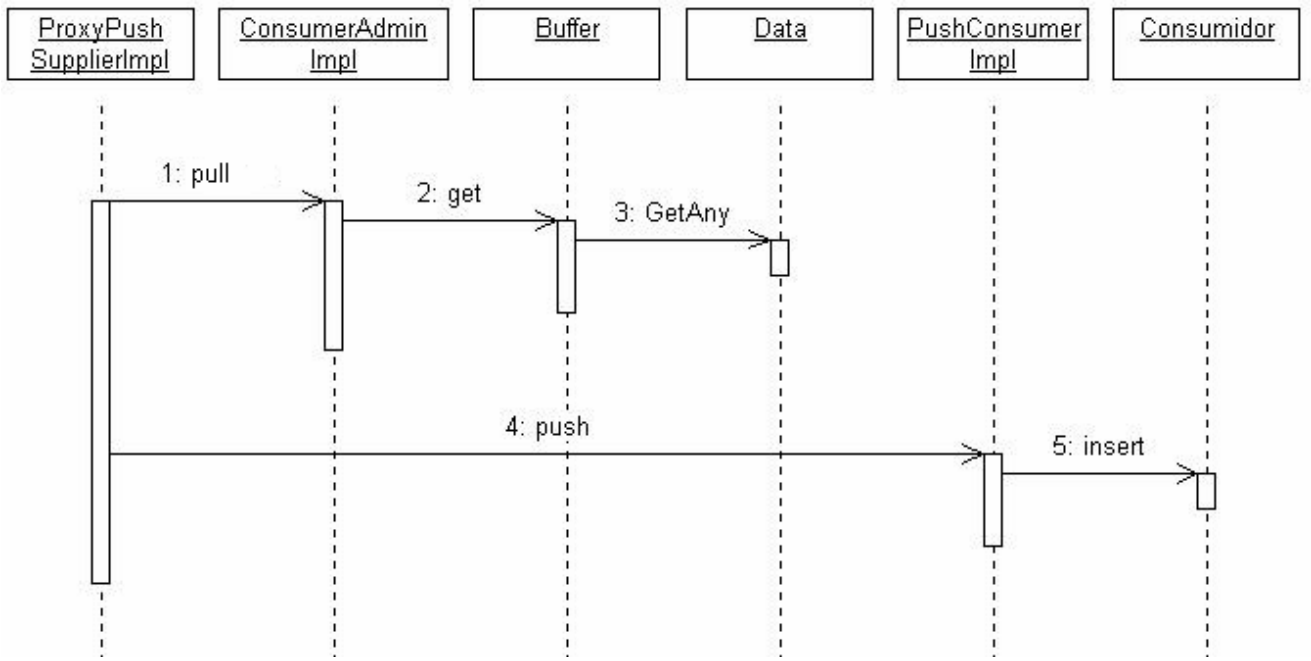
Extracción del canal de un evento por el método Pull

En este caso el PullConsumer invoca el método pull que lo deja bloqueado hasta la llegada de un evento nuevo. El procedimiento es muy parecido al anterior pero no existe la posibilidad de que el objeto Any que llegue al PullConsumer esté vacío.



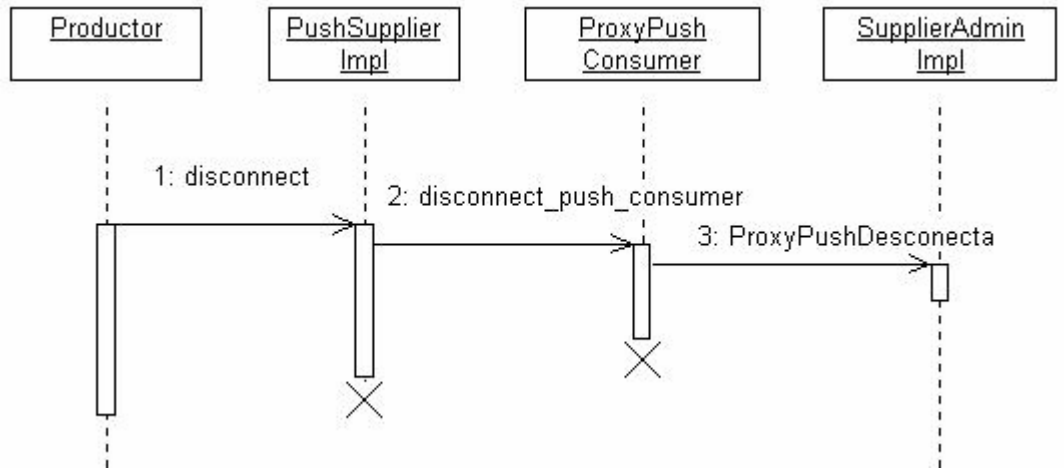
Extracción del canal de un evento por el método Push

En este caso es el propio canal quién envía el evento al PushConsumer. El ProxyPushSupplier realiza pull sobre el administrador de consumidores lo que quiere decir que se bloquea hasta que éste le envíe un evento nuevo. Cuando b tiene lo envía al PushConsumer a través del método push. Una vez con el evento en su poder el PushConsumer lo introduce en el consumidor invocando el método insert.



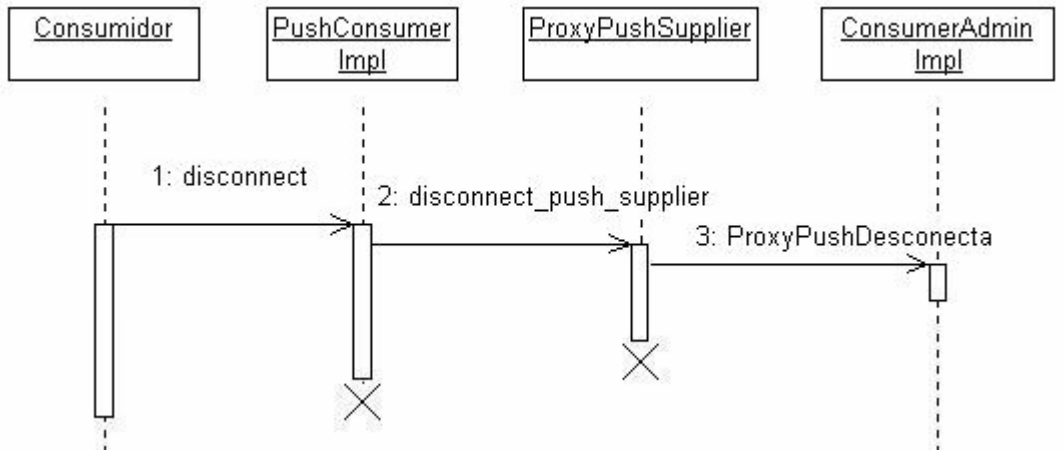
Desconexión de un PushSupplier del canal

Cuando el productor decide eliminar a un PushSupplier invoca el método disconnect, lo que provoca la destrucción de este objeto. Sin embargo, antes de destruirse, el PushSupplier invoca el método disconnect_push_consumer del ProxyPushConsumer quién avisa al administrador de productores de que quiere darse de baja y luego se destruye.



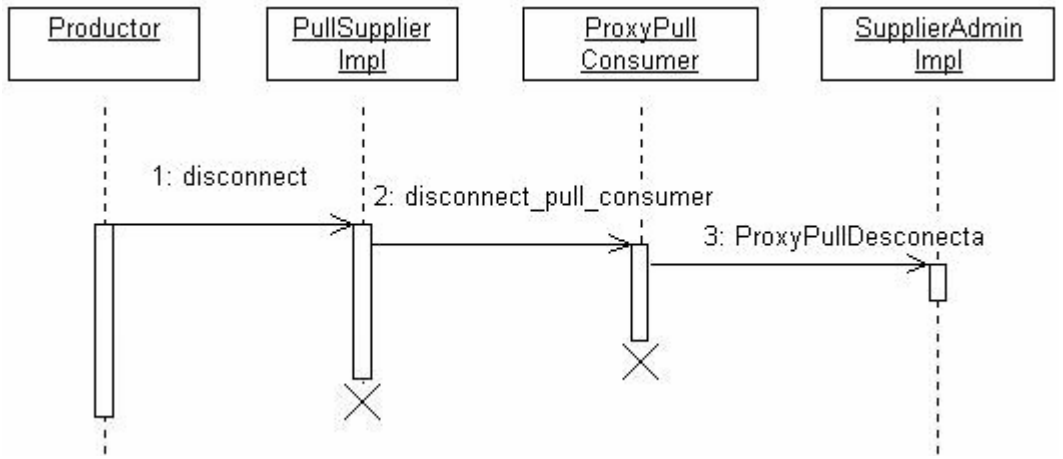
Desconexión de un PushConsumer del canal

Cuando el consumidor decide eliminar a un PushConsumer invoca el método disconnect, lo que provoca la destrucción de este objeto. Sin embargo, antes de destruirse, el PushConsumer invoca el método disconnect_push_supplier del ProxyPushSupplier quién avisa al administrador de consumidores de que quiere darse de baja y luego se destruye.



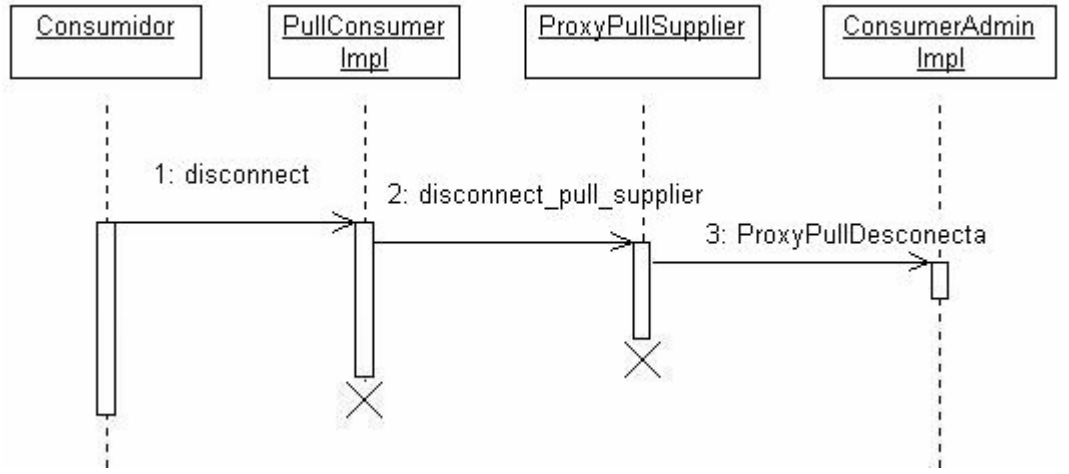
Desconexión de un PullSupplier del canal

Cuando el productor decide eliminar a un PullSupplier invoca el método disconnect, lo que provoca la destrucción de este objeto. Sin embargo, antes de destruirse, el PullSupplier invoca el método disconnect_pull_consumer del ProxyPullConsumer quién avisa al administrador de productores de que quiere darse de baja y luego se destruye.



Desconexión de un PullConsumer del canal

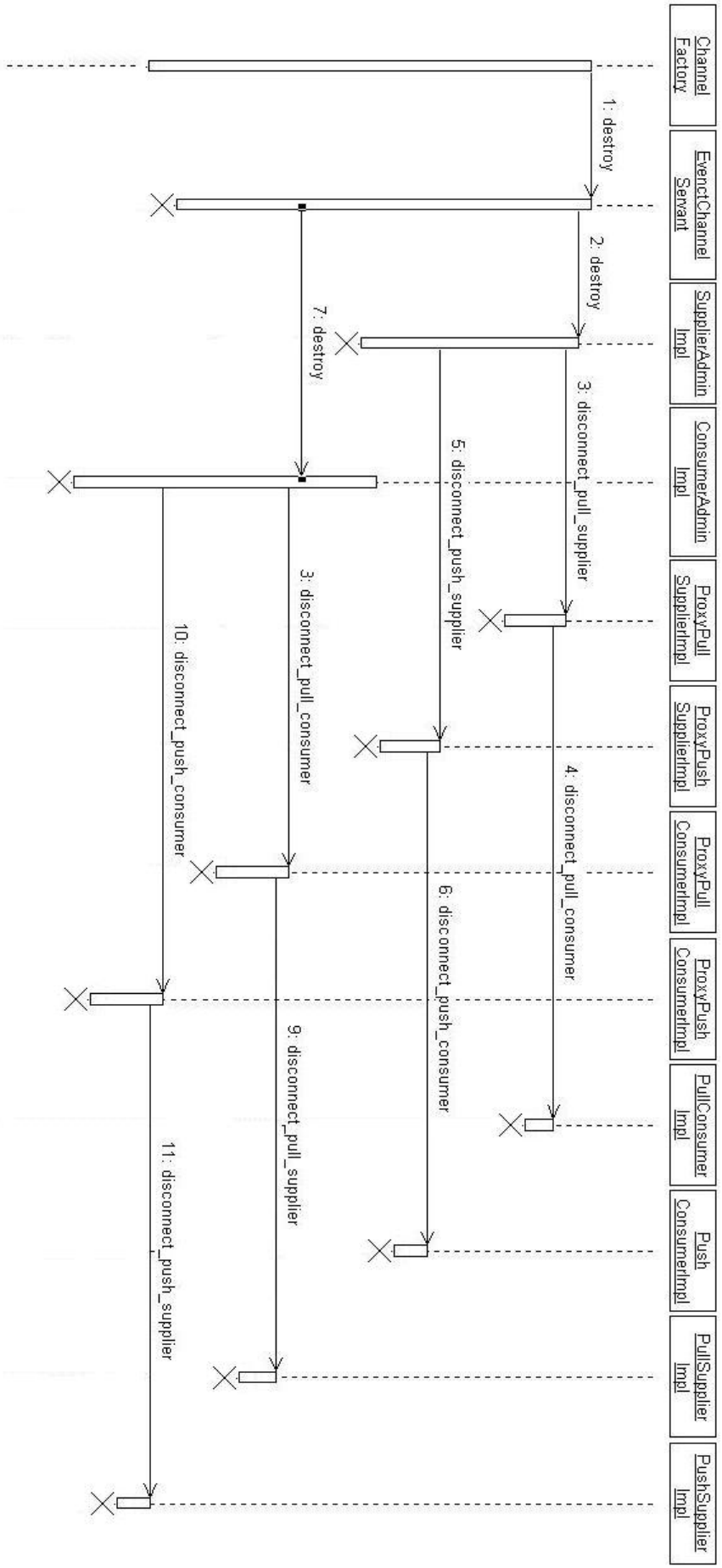
Cuando el consumidor decide eliminar a un PullConsumer invoca el método disconnect, lo que provoca la destrucción de este objeto. Sin embargo, antes de destruirse, el PullConsumer invoca el método disconnect_pull_supplier del ProxyPullSupplier quién avisa al administrador de consumidores de que quiere darse de baja y luego se destruye.



Destrucción del canal

Una vez que la aplicación decide destruir el canal de eventos mediante la orden destroy, el canal transmite esta orden a los administradores quienes la transmiten a su vez a los proxies quienes la transmiten finalmente a los consumidores y productores que hubieran entregado su referencia.

Todos los objetos involucrados en el canal son así destruidos.


ChannelFactory
EventChannelServant
SupplierAdminImpl
ConsumerAdminImpl
ProxyPullSupplierImpl
ProxyPushSupplierImpl
ProxyPullConsumerImpl
ProxyPushConsumerImpl
PullConsumerImpl
PushConsumerImpl
PullSupplierImpl
PushSupplierImpl
1: destroy
2: destroy
7: destroy
3: disconnect_pull_supplier
5: disconnect_push_supplier
3: disconnect_pull_consumer
4: disconnect_pull_consumer
10: disconnect_push_consumer
6: disconnect_push_consumer
9: disconnect_pull_supplier
11: disconnect_push_supplier

7. EventChannelFactory

Este paquete contiene las tres clases que componen la aplicación encargada de crear los canales de eventos.

EventChannelFactory

Clase que contiene el main de la aplicación y que se encarga de instanciar la clase PricipalFrame.

PrincipalFrame

Como su nombre indica, esta clase compone el marco principal de la aplicación. Este marco posee un menú con tres opciones:

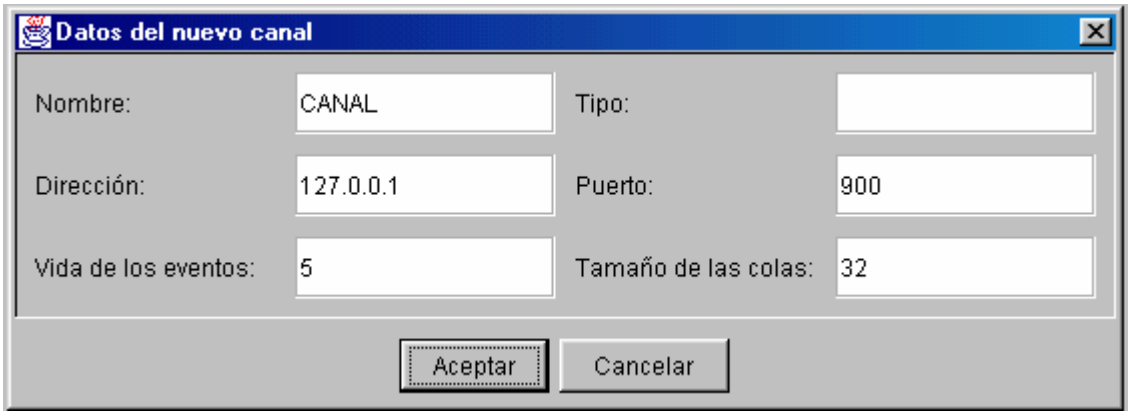
- Crear Canal. Instancia un objeto de la clase Channel.
- Destruir Canal. Destruye el canal que esté seleccionado en ese momento.
- Salir. Sale de la aplicación destruyendo previamente todos los canales que se hubieran creado.

Channel

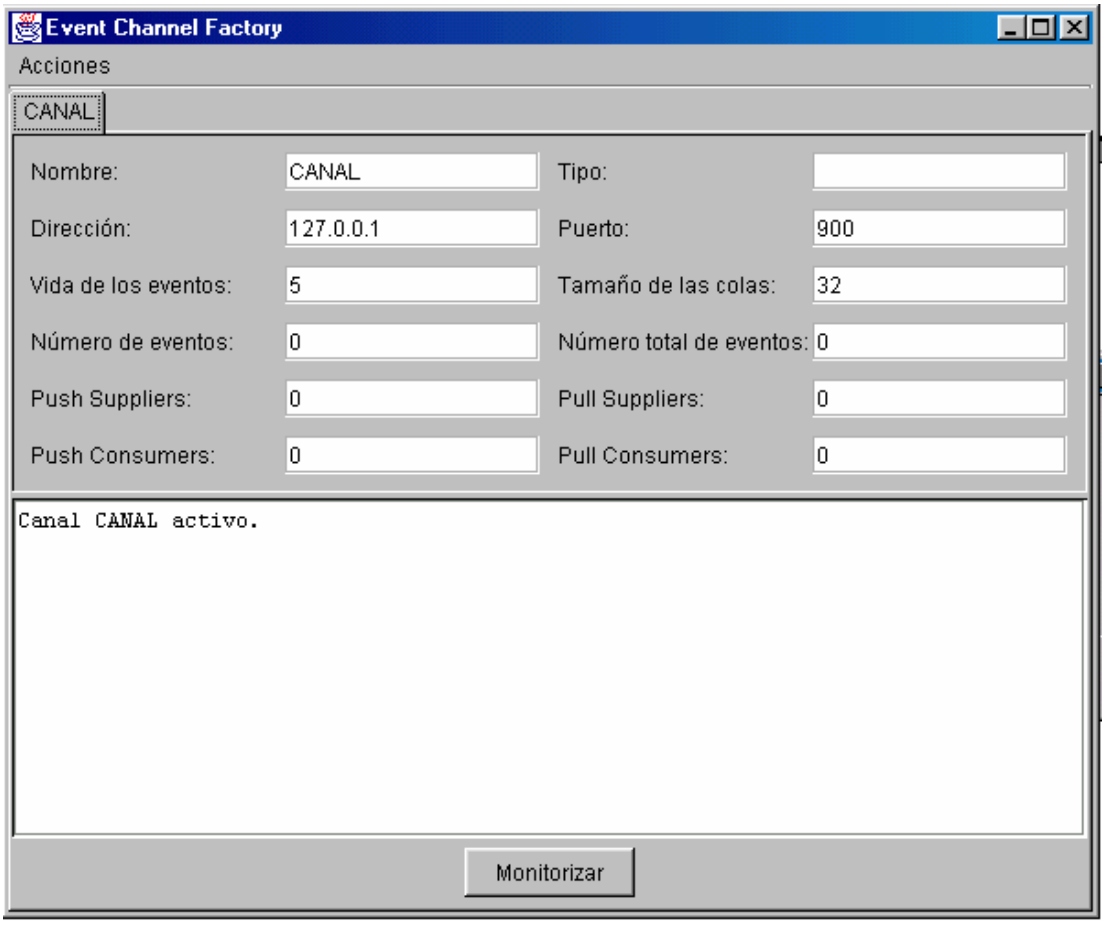
Clase que se encarga de crear un canal nuevo y presentar un entorno gráfico de monitorización del mismo.

En primer lugar se demandan los datos del nuevo canal a crear. Que son:

- Nombre del canal a crear (con el que se dará de alta en el servicio de nombrado).
- Tipo del canal a crear (contexto en el servicio de nombrado).
- Dirección IP de la máquina donde corre el servidor de nombres.
- Puerto de escucha del servidor de nombres.
- Vida de los eventos. Tiempo transcurrido el cual los eventos son eliminados.
- Tamaño de la cola del ProxyPushConsumer.



Una vez aceptados estos datos, se crea un canal y se le da de alta en el servidor de nombres especificado. Este canal está listo para recibir eventos. La pantalla cambia a otra que permite monitorizar el canal.



En este panel, tenemos los datos que se introdujeron anteriormente y que caracterizan al canal además de algunos otros:

- Número de PushSuppliers conectados al canal.
- Número de PullSuppliers conectados al canal.
- Número de PushConsumers conectados al canal.
- Número de PullConsumers conectados al canal.
- Número total de eventos gestionados por el canal desde su creación.
- Número de eventos que almacena actualmente el canal.
- Panel de texto editable donde se muestran los mensajes de información y error generados por el canal.

8. ClientsFactory

Este paquete contiene las seis clases que componen la aplicación encargada de crear clientes para el canal de eventos.

ClientsFactory

Clase que contiene el main de la aplicación y que se encarga de instanciar la clase PricipalFrame.

PrincipalFrame

Como su nombre indica, esta clase compone el marco principal de la aplicación. Este marco posee un menú con seis opciones:

- Crear PushSupplier. Crea un PushSupplier e instancia un objeto de la clase PushSupplierPanel.
- Crear PullSupplier. Crea un PullSupplier e instancia un objeto de la clase PullSupplierPanel.
- Crear PushConsumer. Crea un PushConsumer e instancia un objeto de la clase PushConsumerPanel.
- Crear PullConsumer. Crea un PullConsumer e instancia un objeto de la clase PullConsumerPanel.
- Destruir. Destruye el cliente que esté seleccionado en ese momento.
- Salir. Sale de la aplicación destruyendo previamente todos los clientes que se hubieran creado.

PushSupplierPanel

Clase que se encarga de conectar el nuevo PushSupplier a un canal y presentar un entorno gráfico de monitorización del mismo.

En primer lugar se demandan los datos siguientes:

- Nombre del canal al que conectarse.
- Tipo del canal al que conectarse.
- Dirección IP de la máquina donde corre el servidor de nombres.
- Puerto de escucha del servidor de nombres.
- Envío o no de la referencia del PushSupplier al ProxyPushConsumer correspondiente.

El panel presenta cuatro botones con las funciones:

- Conectar el PushSupplier al canal.
- Desconectar el PushSupplier del canal.
- Enviar un evento individual consistente en un vector que incluye un String con el valor "Push", un Interger con valor cero y un objeto Date con la fecha de generación del evento.
- Enviar una ráfaga de 2000 eventos de forma consecutiva. Los eventos son iguales a los individuales salvo que el Integer en lugar de valer cero tiene el valor de la posición del evento concreto en la ráfaga.

Además se dispone de un contador de eventos enviados que se resetea al desconectar el PushSupplier y de un panel de texto editable donde se muestran los mensajes de información y error.

PushConsumerPanel

Clase que se encarga de conectar el nuevo PushConsumer a un canal y presentar un entorno gráfico de monitorización del mismo.

En primer lugar se demandan los datos siguientes:

- Nombre del canal al que conectarse.
- Tipo del canal al que conectarse.
- Dirección IP de la máquina donde corre el servidor de nombres.
- Puerto de escucha del servidor de nombres.

El panel presenta dos botones con las funciones:

- Conectar el PushSupplier al canal.
- Desconectar el PushSupplier del canal.

Además se dispone de un contador de eventos recibidos que se resetea al desconectar el PushConsumer y de un panel de texto editable donde se muestran

los mensajes de información y error (incluyendo un mensaje cada vez que se recibe un nuevo evento con información sobre el mismo).

PullSupplierPanel

Clase que se encarga de conectar el nuevo PullSupplier a un canal y presentar un entorno gráfico de monitorización del mismo.

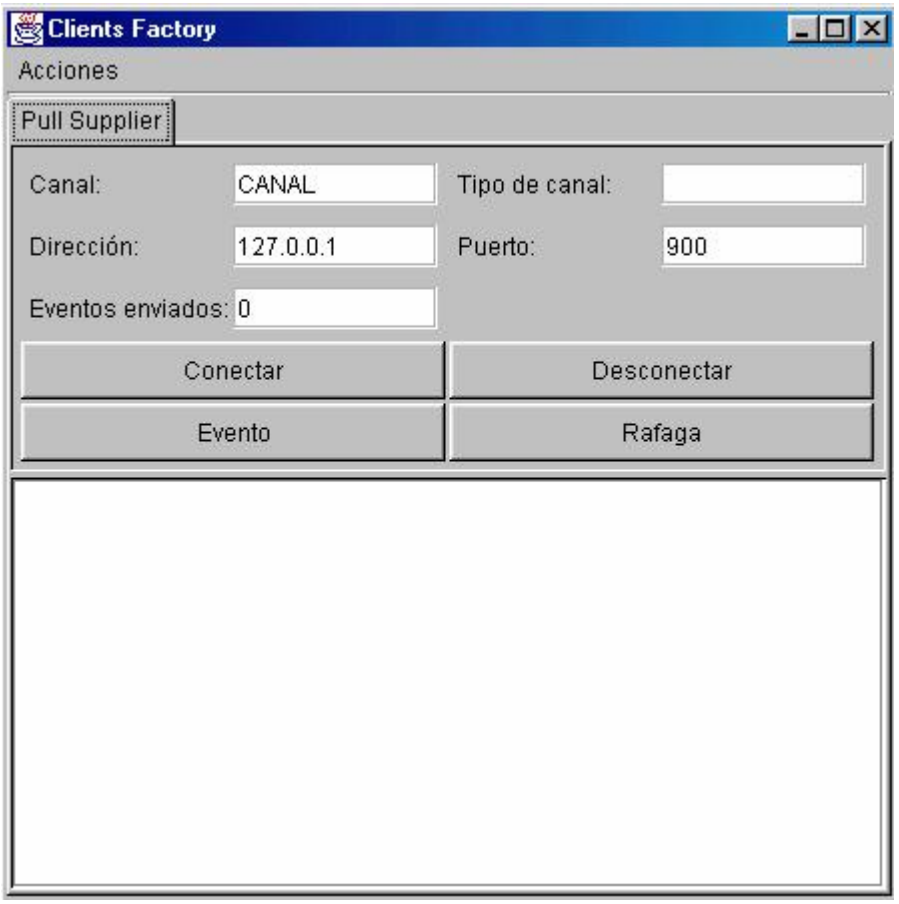
En primer lugar se demandan los datos siguientes:

- Nombre del canal al que conectarse.
- Tipo del canal al que conectarse.
- Dirección IP de la máquina donde corre el servidor de nombres.
- Puerto de escucha del servidor de nombres.

El panel presenta cuatro botones con las funciones:

- Conectar el PullSupplier al canal.
- Desconectar el PullSupplier del canal.
- Generar un evento individual consistente en un vector que incluye un String con el valor "Pull", un Interger con valor cero y un objeto Date con la fecha de generación del evento.
- Enviar una ráfaga de 2000 eventos de forma consecutiva. Los eventos son iguales a los individuales salvo que el Integer en lugar de valer cero tiene el valor de la posición del evento concreto en la ráfaga.

Además se dispone de un contador de eventos enviados que se resetea al desconectar el PullSupplier y de un panel de texto editable donde se muestran los mensajes de información y error.



PullConsumerPanel

Clase que se encarga de conectar el nuevo PullConsumer a un canal y presentar un entorno gráfico de monitorización del mismo.

En primer lugar se demandan los datos siguientes:

- Nombre del canal al que conectarse.
- Tipo del canal al que conectarse.
- Dirección IP de la máquina donde corre el servidor de nombres.
- Puerto de escucha del servidor de nombres.
- Envío o no de la referencia del PullConsumer al ProxyPullSupplier correspondiente.

El panel presenta cuatro botones con las funciones:

- Conectar el PullConsumer al canal.
- Desconectar el PullConsumer del canal.

- Petición de un evento al canal con el método pull.
- Petición de un evento al canal con el método try_pull.

Además se dispone de un contador de eventos recibidos que se resetea al desconectar el PullConsumer y de un panel de texto editable donde se muestran los mensajes de información y error (incluyendo un mensaje cada vez que se recibe un nuevo evento con información sobre el mismo).

CAPÍTULO 6:

CONCLUSIONES

CONCLUSIONES

1. Historia del proyecto

La realización del proyecto puede dividirse en fases que, aunque están claramente definidas, no siempre han estado temporalmente separadas; pues ha sido algo común la realización de tareas en paralelo e incluso la inclusión de ciclos iterativos de desarrollo.

El primer paso consistió en familiarizarme con la arquitectura CORBA y sus características insistiendo claro está en la definición del servicio de eventos. En paralelo a esta tarea comencé a aprender Java pues desconocía este lenguaje. El conjunto de estas labores ocupó aproximadamente un mes y medio.

Al final del primer mes comencé a plantear la estructura del proyecto estableciendo las clases principales y sobre todo el esqueleto de interfaces a implementar. Durante un mes y medio diseñé la aplicación estableciendo el definitivo diagrama de clases.

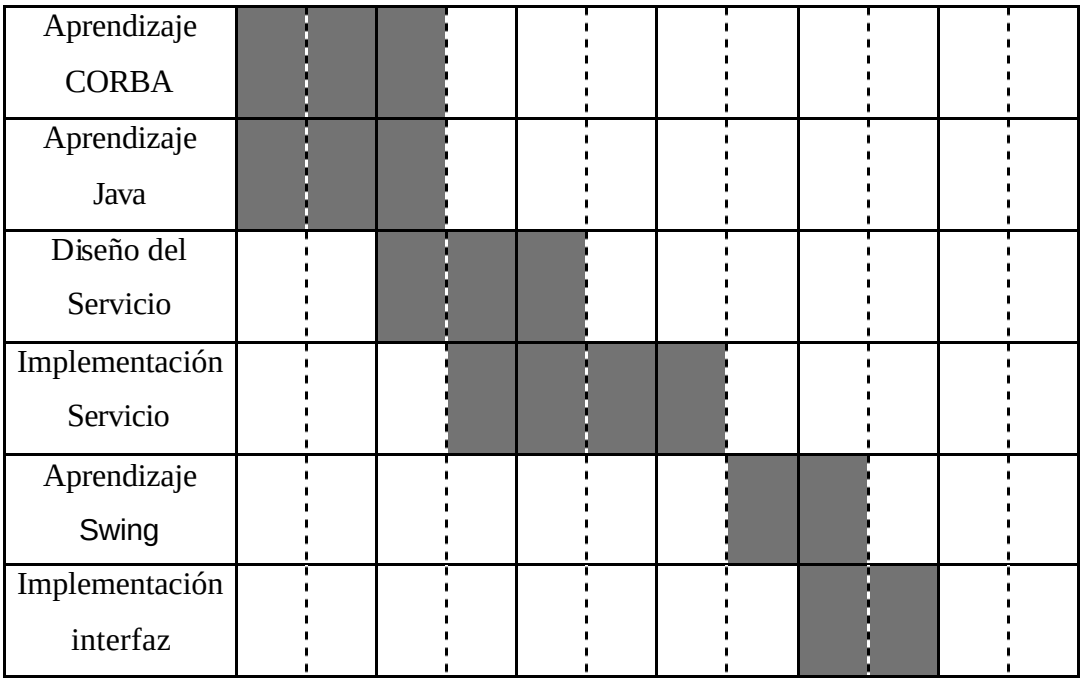
Mientras diseñaba también comencé a escribir código hasta establecer una primera base compilable muy básica pero que, sin embargo, contemplaba todas las funcionalidades.

Al cabo de dos meses el servicio estaba terminado pero los programas cliente que debían probarlo, aunque tenían la misma funcionalidad que en la actualidad, estaban desarrollados en modo consola.

Durante un mes y medio estudié las clases del paquete Swing de Java y transformé la interfaz de usuario de los programas cliente hasta dejarlas como anteriormente se mostró.

Por último, durante el último mes recopilé y di formato a todas las notas que había ido tomando durante el proceso documentando todo el proyecto. Antes de terminar realicé de nuevo todas las pruebas que ya había realizado para confirmar que no quedaba ningún cabo suelto.

En la siguiente tabla se refleja la evolución temporal del proyecto en sus diferentes fases:



Documentación y pruebas						
	Mes 1	Mes 2	Mes 3	Mes 4	Mes 5	Mes 6

2. Pruebas del producto

Las pruebas a las que se ha sometido el proyecto se han realizado gracias a las aplicaciones que se incluyen y se han producido tanto en red real como virtual.

- Creación y destrucción de canales.
- Creación y destrucción de todos los clientes.
- Conexión repetida de todos los clientes
- Desconexión repetida de todos los clientes.
- Introducción de eventos tipo Push de forma individual.
- Introducción de eventos tipo Pull de forma individual.
- Introducción de ráfagas de 2000 eventos tipo Push consecutivos.
- Introducción de ráfagas de 2000 eventos tipo Pull consecutivos.
- Extracción de eventos tipo Push manteniendo el orden de llegada.
- Extracción de eventos tipo Pull manteniendo el orden de llegada.
- Extracción de eventos tipo TryPull manteniendo el orden de llegada.
- Interconexión de todas las combinaciones de clientes al canal.
- Eliminación de eventos transcurrido el tiempo de vida de éstos.
- Repetición de las pruebas anteriores sin posibilidad de localizar el servicio de nombrado, el canal o los clientes.

El resultado de las pruebas ha sido satisfactorio en todos los casos, recibiendo los mensajes de error pertinentes en las zonas de texto de las aplicaciones.

3. Conclusiones

Para concluir queda decir que se han alcanzado los objetivos propuestos al comenzar el proyecto:

- Aprendizaje de un lenguaje de programación de alto nivel como es Java
- Estudio y comprensión de la estructura distribuida de objetos propuesta por CORBA.
- Implementación del servicio de eventos de CORBA para Java IDL utilizando el servicio de nombrado para localizar los canales.
- Creación de un entorno de pruebas amable que permite monitorizar tanto el canal como los clientes.

4. Líneas de continuación

Una posible línea de continuación es agregar al canal la posibilidad de trabajar con eventos tipados. En este tipo de comunicación los productores invocan operaciones en los productores utilizando una interfaz I previamente definida en IDL. Las operaciones que podrá contener esta interfaz cumplen:

- Todos los parámetros son de tipo in.
- No se permite devolver valores por parte de los métodos.

Otra posible mejora al servicio es el establecimiento de combinaciones de canales que permitan un enrutado de los eventos de manera que no todos los eventos que se produzcan lleguen a todos lo consumidores sino que se permitan filtrados.

Un campo en estudio y expansión constante el de envío de eventos en tiempo real. Para dotar al canal de eventos de esta capacidad es necesario dotarlo con:

- Políticas de programación en tiempo real.
- Reparto en tiempo real. Es decir, respetando un orden de prioridades.
- Implementación de interfaces con una QoS conocida y aceptada entre los distintos clientes.
- Estrategias de concurrencia flexibles.
- Correlación y filtrado de eventos.

5. *Bibliografía*

- "Java. Guía de desarrollo"

  Jamie Jaworski

  Prentice Hall 1997

- "The Unified Modeling Language User Guide"

  Grady Booch

  James Rumbaugh

  Ivar Jacobson

  Addison Wesley 1999

- "Modelado de objetos con UML"

  Pierre-Alain Muller

  Eyrolles 1997

- "The Java tutorial".

  M. Campione y K.Walrath.

  Documento obtenido en web:

  http://www.java.sun.com/docs/books/tutorial/index.html.

- "Tutorial de Java"

  Agustín Froufe

  Documento obtenido en web:

  http://members.es.tripod.de/froufe

- Revista "Solo programadores" números 68y 69.

- "Java Development Kit 1.3. Documentation"

  Documentación disponible en la web:

  http://www.java.sun.com/products/jdk/1.2/docs/index.html.

- "The CORBA 2.0 Specification"

  Documentación disponible en la web:

  http://www.omg.org