Un *framework* para persistencia en CORBA

Diego Sevilla Ruiz[1], Jesús J. García Molina[2], Marcos Menárguez Tortosa[2]

[1] Departamento de Ingeniería y Tecnología de Computadores
[2] Departamento de Lenguajes y Sistemas Informáticos
Universidad de Murcia
Campus de Espinardo
30080 MURCIA
{dsevilla, jmolina}@um.es, marcos@dif.um.es

Resumen Desde la aparición de CORBA, el soporte de persistencia a objetos implementación ha sido problemático, primero por la vaga especificación del ciclo de vida de los objetos dada por el BOA (*Basic Object Adapter*) y después por la deficiente calidad de la especificación del POS (*Persistent Object Service*), lo que lleva a menudo a implementaciones *ad hoc* e incompatibles. En este trabajo presentamos un *framework* que facilita un soporte de persistencia de forma automática a objetos CORBA. El soporte es transparente a los clientes, que pueden utilizar los objetos persistentes a través de interfaces factoría e iterador, está basado sólo en estándares y contempla una solución para el problema de las referencias persistentes en el contexto del BOA, además de contemplar el uso del reciente POA (*Portable Object Adapter*).
Palabras clave: CORBA, Objetos Persistentes, *Framework* para persistencia.

1 Introducción

CORBA [1] se ha erigido como uno de los estándares más consolidados para la implementación de aplicaciones distribuidas. La necesidad de almacenamiento persistente es frecuente en la implementación de aplicaciones de esta naturaleza. Es claro que la *persistencia* es una cuestión que concierne a los objetos servidores que pueden requerir un almacenamiento persistente para mantener su estado.

CORBA fue diseñado, desde un principio, con el objetivo principal de estandarizar la parte cliente de la arquitectura y adoleció de deficiencias en su soporte del servidor, haciendo que las implementaciones de objetos servidores (objetos implementación) CORBA fueran incompatibles entre ORBs de distintos fabricantes. A esto contribuyó una especificación incompleta del BOA (el adaptador de objetos) que llevó a los fabricantes a ofrecer soluciones propietarias en la parte servidor, y por tanto a la incompatibilidad de las implementaciones de objetos servidores. Así, siendo la implementación de objetos servidores uno de los puntos de mayor disputa de CORBA, el soporte de estado persistente para objetos CORBA ha sido siempre un caballo de batalla para el OMG (*Object Management Group*).

En los últimos años se han desarrollado diversos *framework* [17] que proporcionan a las aplicaciones orientadas a objetos acceso transparente a sistemas de bases de datos relacionales. Hace un año iniciamos un proyecto [2] para el desarrollo de un *framework* de este tipo para el caso de aplicaciones distribuidas basadas en CORBA. En este trabajo presentamos el estado actual de este proyecto. Este *framework* intenta suplir las deficiencias que en la parte servidor presenta CORBA, ofreciendo además una interfaz de acceso a los clientes independiente de si los objetos servidores son persistentes o no, y generando automáticamente el soporte necesario para la persistencia.

Este trabajo se organiza de la siguiente manera: En la sección 2 se discuten las alternativas existentes para persistencia de objetos CORBA, introduciendo los requisitos que deben ser satisfechos. En la sección 3 presentamos, a través de un ejemplo sencillo, el modelo de trabajo definido por el *framework*, la visión que se presenta a los clientes de los objetos persistentes y las decisiones que guiaron la implementación misma del *framework* y del soporte de persistencia para objetos servidores. Por último, la sección 4 presenta las conclusiones y el trabajo futuro.

2 Persistencia en CORBA: Problemas y requisitos

El OMG [3] propuso el *Persistent Object Service* (POS) [4] que nunca fue implementado, aunque sí muy criticado [5], debido en parte al equívoco soporte de servidor ofrecido por el BOA, y en parte a la filosofía misma del POS, que obligaba a los clientes a manejar de forma explícita a los objetos persistentes. Las casas comerciales, presionadas por la demanda, comenzaron a ofrecer soportes propietarios [6,7], que no hacían más que incrementar la incompatibilidad entre las implementaciones de objetos servidor con soporte de persistencia.

Esto llevó al OMG a lanzar una petición de propuestas (*Request For Proposals, RFP*) [8] para un servicio de persistencia (el *Persistent State Service 2.0*, PSS) que solucionara estos inconvenientes, y de la cual ya se han obtenido propuestas [11], aunque el OMG todavía no la ha aceptado definitivamente, por lo que no existen implementaciones maduras disponibles.

Debido a esta situación, los desarrolladores de aplicaciones CORBA con alguna necesidad de almacenamiento persistente se ven obligados a implementar soportes de persistencia *ad hoc*, que ofrecen una funcionalidad incompleta (por ejemplo, se obvia el *caching* de objetos), a menudo no portables ni reutilizables.

Para resolver estos problemas, iniciamos el desarrollo de un *framework* que genera automáticamente el soporte necesario para dotar de persistencia a objetos CORBA, que se diferencia del PSS en las siguientes características:

- Mientras que el PSS separa la definición del estado persistente del interfaz ofrecido a los clientes, es habitual que los objetos persistentes hagan público su estado en su interfaz. Nuestro *framework* automatiza este proceso;
- el estado persistente y las clases necesarias para su manejo se generan de forma automática y sólo es necesario el uso de IDL (*Interface Definition Language*);

– automatiza la construcción de *wrappers* CORBA de bases de datos relacionales existentes a través de un "*font end*" que interpreta la estructura de tablas;
– no es un servicio de persistencia, sino un *framework*, por lo que es extensible;

Requisitos de la solución

El primer paso para desarrollar una solución a los problemas señalados fue el establecimiento de una serie de requisitos que ésta debería cumplir. Así, establecimos los siguientes:

1. **100% CORBA**: Deberá estar basada únicamente en el estándar CORBA. Un mecanismo tan general como un *framework* de persistencia no puede estar ligado a ninguna característica puntual, y por ello incompatible, de un producto particular.
2. **Transparencia**: Deberá ofrecer a los clientes un acceso a los objetos que ofrezca *transparencia de persistencia*, esto es, que sea independiente de si éstos son persistentes o no (*ortogonal*).
3. **Correspondencia IDL/SQL**: Deberá ofrecer un mecanismo para definir qué objetos son persistentes y qué atributos poseen esos objetos, permitir la definición de objetos compuestos a través de las relaciones de asociación y agregación, así como permitir relaciones de herencia. También deberá: generar definiciones IDL para los objetos persistentes que se puedan integrar con el resto de definiciones IDL del sistema, ofrecer una correspondencia entre tipos IDL y tipos de tablas relacionales (SQL), así como el código para recuperar y guardar el estado de los objetos en la base de datos.
4. **Eficiencia**: Deberá poseer una eficiencia *razonable*, lo que obliga, entre otras cosas, a implementar mecanismos de *caché* de objetos, decidir en tiempo de ejecución qué objetos serán *pasivados*[1] (introducidos en la base de datos) y qué otros serán *activados* (extraídos de la base de datos), implementar estrategias de acceso a los objetos que permitan explotar el patrón de accesos subyacente (como iteradores, etc.)
5. **Soporte de transacciones**: El *framework* deberá permitir que el cliente pueda establecer los límites de las transacciones, asegurando la atomicidad de las mismas, y con operaciones `commit` y `rollback`.

3 Un *framework* para persistencia en CORBA

En este momento, el *framework* consta de un generador de implementaciones de objetos persistentes a partir de sus especificaciones y de un conjunto de clases que ofrecen soporte a las clases específicas construidas por el generador para cada tipo de objetos, ofreciendo soporte de almacenamiento, gestión de caché y activación bajo demanda. A continuación, describiremos la estructura del *framework*.

[1] Hemos utilizado *pasivar* para hacer referencia al verbo inglés *passivate*, común en la bibliografía sobre CORBA y persistencia.

3.1 Generación de objetos persistentes

Nuestro *framework* permite la definición de objetos persistentes bien a partir de DDL (*Data Definition Language*) o bien de SQL, que se traduce a especificaciones DDL. El uso de SQL permite la integración automática de bases de datos heredadas (*legacy*).

Aunque la propuesta para el PSS [11] define un nuevo lenguaje de especificación de estados de objetos persistentes llamado PSDL (*Persistent State Definition Language*), un superconjunto de IDL que introduce ciertas construcciones para reflejar la estructura y características del estado persistente de los objetos que lo necesiten, consideramos a DDL (definido en el POS [4]) más conveniente que PSDL por varias razones: (1) DDL, al ser un subconjunto de IDL, permite que el análisis sintáctico y semántico se pueda realizar con herramientas de análisis de ficheros IDL, de gran disponibilidad actualmente; (2) los objetos definidos en DDL se pueden incluir en el conjunto de definiciones IDL que forman el interfaz de la aplicación CORBA de forma automática, y (3) PSDL introduce nuevas construcciones que añaden una complejidad a la especificación y que son, en la mayoría de los casos, equivalentes a las presentes en DDL (por ejemplo, la construcción PSDL `storagetype` se puede ver como equivalente a la construcción `interface` de DDL, y la construcción `storagehome` a la factoría creada de forma implícita por el *framework*). Como una restricción adicional a DDL, consideramos *clave* del objeto aquellos atributos que se definieran con el calificativo "`readonly`".

La Figura 2 esquematiza el proceso de traducción de una especificación DDL, generándose los siguientes ficheros: (1) **Un fichero SQL**, que contiene las sentencias "`create table`" necesarias para dar soporte al almacenamiento en la base de datos de los objetos persistentes; (2) **Un fichero IDL**, con las definiciones de las interfaces contenidas en el fichero DDL *más* aquellas que son necesarias para implementar el acceso transparente a la base de datos; (3) **Ficheros implementación (en Java) de las interfaces definidas en el fichero IDL**.

Así, el proceso a seguir para utilizar el *framework* pasaría por escribir las definiciones de objetos persistentes utilizando DDL, ejecutar el Generador, importar las definiciones SQL en la base de datos, y finalmente compilar los ficheros IDL y los ficheros de implementación generados.

Para ilustrar el uso de DDL y cómo el *framework* irá generando las clases necesarias, comenzaremos introduciendo un sencillo ejemplo de una biblioteca. En ella, se quiere tener registro de los libros, sus autores y las editoriales. Consideramos el nombre de un libro como su clave. Utilizando DDL, la definición podría quedar (en el fichero `biblioteca.ddl`) como se ve en la Figura 1.

3.2 La visión del cliente

La persistencia es una cuestión a tener en cuenta en la parte servidor de la interacción Cliente/Servidor proporcionada por CORBA. Así, los clientes deberán acceder a los objetos descritos en DDL a través de una interfaz que les permita

```
// DDL
module biblioteca {
    interface autor {
        readonly attribute string nombre;
        readonly attribute string apellidos;
        attribute short edad;
    };
    interface editorial {
        readonly attribute string nombre;
        attribute string direccion;
    };
    interface libro {
        readonly attribute string nombre;
        attribute autor autorLibro;
        attribute editorial editor;
        attribute long año;
    };
};
```

Figura 1. Definición DDL de nuestro ejemplo.

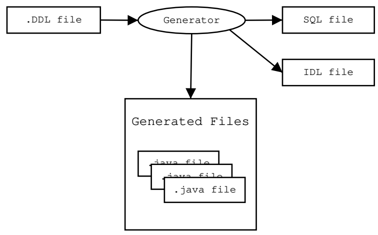

Figura 2. Flujo de datos del generador de implementaciones de objetos persistentes.

manejar el conjunto de objetos de forma *independiente* del carácter persistente de los mismos. Dos patrones de diseño son comunes en el diseño de servicios CORBA: las *factorías* y los *iteradores* [12], como se puede ver en la especificación de los servicios estándar de CORBA [4]. Las factorías encapsulan la gestión del ciclo de vida de los objetos persistentes. Los iteradores aportan una forma conveniente de recorrer y tratar de forma ordenada el conjunto de objetos manejados por una factoría.

Como vemos en la Figura 2, el Generador crea un fichero IDL que incluye al fichero DDL junto con una clase "factoría" y otra "iterador" para cada interfaz de objeto persistente definida en el fichero DDL. En el ejemplo anterior, el conjunto de interfaces generadas (en el fichero `biblioteca_gen.idl`) para `libro` sería:

```
// IDL
#include "biblioteca.ddl"
module biblioteca {
    typedef sequence<biblioteca::libro> libroList;
    interface libroIterator {
        biblioteca::libro next();
        libroList next_n(in long howMany);
    };
    interface libroFactory {
        biblioteca::libro getObject(in string nombre);
        void remove(in string nombre);
        boolean exists(in string nombre);
        libroList list( in string condition, in long howMany, out libroIterator it);
    };
};
```

En nuestro diseño, cada factoría maneja al conjunto de objetos de un tipo DDL específico, permitiendo recuperar (o crear, si no existe) un objeto dada su clave (operación `getObject`), eliminar un objeto del conjunto (`remove`), preguntar por la existencia de un objeto (`exists`) y listar el conjunto de objetos (`list`). Este último método hace uso de la clase iterador creada para cada tipo de objeto persistente. Las clases iterador ofrecen dos métodos: `next`, que retorna el siguiente elemento de la colección, y `next_n`, que retorna una lista de n elementos como máximo. Esta última operación permite adaptar la eficiencia de la aplicación a las características del entorno (como la velocidad y tasa de transferencia de los enlaces), y es similar a la existente en la interfaz `CosNaming::BindingIterator` definida en el *Naming Service* de CORBA [4].

Nótese cómo los distintos métodos de selección de un objeto de entre los del conjunto utilizan el grupo de atributos a los que se le ha dado la agrupación semántica de *clave*. Mientras que un `libro` queda completamente identificado (en nuestro diseño) por su nombre, un `autor` lo es por su nombre y sus apellidos.

Finalmente, un cliente puede usar estas clases siguiendo el patrón habitual para factorías, esto es, obtener la referencia a la factoría a través de algún servicio de publicación (referencias iniciales, servicio de nombres, *trading*, etc.)[2], utilizar el método `getObject` sobre la factoría y a continuación manipular los datos del objeto retornado. Se muestra el uso en Java y C++:

Java	**C++**
`import biblioteca.*;`	`using namespace biblioteca;`
`// ...`	`// ...`
`libroFactory lf= // ref.`	`libroFactory_var lf= // ref.`
`// Obtener “Design Patterns”`	`// Obtener “Design Patterns”`
`libro miLibro;`	`libro_var miLibro;`
`miLibro = lf.getObject("D. P.");`	`miLibro = lf->getObject("D. P.");`
`// Operar con miLibro imprimir`	`// Operar con miLibro imprimir`
`// año de publicación:`	`// año de publicación:`
`System.out.println(miLibro.año());`	`cout << miLibro->año() << endl;`
`// Y también modificarlo:`	`// Y también modificarlo:`
`autorFactory af= // Ref.`	`autorFactory_var af= // Ref.`
`miLibro.autor(`	`miLibro->autor(`
` af.getObject("Gamma, E.");`	` af->getObject("Gamma, E.");`

Según la definición de la operación `getObject`, ésta crea implícitamente un nuevo objeto `autor` si el autor especificado no existiera. Queda bajo la responsabilidad del cliente establecer previamente de forma conveniente los distintos atributos del (en principio nuevo) objeto `autor`. Esto resalta una de las características más importantes: el cliente es el responsable de asegurar la integridad referencial y las cardinalidades relativas entre los distintos objetos. Otra alternativa sería añadir a DDL construcciones similares a las que aparecen en SQL como `on delete cascade`. Esta alternativa contaría con más inconvenientes que ventajas, ya que (1) resultaría casi imposible, para un sistema de tamaño medio, poder describir las relaciones existentes entre los distintos objetos, obligando así

[2] En nuestra implementación utilizamos el servicio de nombres.

a la inclusión de un número elevado de construcciones distintas de diferenciación muy sutil, y (2) aumentaría la complejidad de forma innecesaria, habida cuenta de que el esquema actual ofrece al usuario una flexibilidad total.

3.3 La implementación

Para la implementación utilizamos el lenguaje Java [13]. Como soporte de base de datos, utilizamos JDBC [14], lo cual nos daba la posibilidad de utilizar un gran número de SGBD destino: finalmente utilizamos *Postgresql* bajo Linux.

Como se muestra en la Figura 2, el Generador, a partir del fichero DDL, debe crear, además del fichero IDL, (1) el fichero con las sentencias SQL [15] que definen las tablas en las que se albergará el estado de los objetos persistentes, y (2) el conjunto de clases que, junto con las provistas por el *framework de persistencia* (en el paquete `dbcollection`), hacen posible que el estado de estos objetos sea persistente.

Correspondencia entre objetos y tablas SQL El fichero SQL incluye una sentencia de creación de tabla para cada interfaz definida en el fichero DDL. Para evitar colisiones de nombres, los nombres de las tablas contienen el nombre completo de la interfaz, incluyendo el nombre de los módulos donde están incluidos. Para nuestro ejemplo sencillo, el fichero SQL producido por el Generador sería el siguiente (se muestra sólo la tabla correspondiente a `libro`):

```
CREATE TABLE biblioteca_libro (
    nombre varchar, autorLibro_nombre varchar,
    autorLibro_apellidos varchar, editor_nombre varchar, año int,
    PRIMARY KEY (nombre),
    FOREIGN KEY (autorLibro_nombre) REFERENCES biblioteca_autor(nombre),
    FOREIGN KEY (autorLibro_apellidos) REFERENCES biblioteca_autor(apellidos),
    FOREIGN KEY (editor_nombre) REFERENCES biblioteca_editorial(nombre)
    );
```

Las asociaciones (por ejemplo, un `libro` tiene un `autor` y un `editor`) se modelan incluyendo tantos atributos en la tabla contenedora como atributos tiene la clave de cada uno de los objetos de los interfaces incluidos.

Al no contemplar en la implementación actual el uso de atributos multivaluados, la correspondencia entre tipos IDL y SQL resulta sencilla.

Las clases generadas El conjunto de clases generadas por el Generador proporcionará la implementación de todas las interfaces que se ofrecen al cliente, así como otras, ocultas a los clientes, que interactúen con el resto del *framework*. Así, dado un interfaz, por ejemplo `libro`, definido en DDL, se generan las siguientes clases (Figura 3):

– *`libro`*`Impl`: Implementa la interfaz CORBA `libro`, proporcionando los métodos de acceso y modificación (*accessor*/*mutator*) para cada uno de los atributos del objeto.

- *libro*`FactoryImpl`: Implementa la factoría para el tipo de objetos persistentes definido por `libro`.
- *libro*`IteratorImpl`: Implementa el mecanismo de iteración a través de una lista de objetos retornada por una factoría.
- *libro*`OA`: El adaptador de objetos para esa interfaz. Esta clase implementa el acceso a bajo nivel a través de sentencias de la base de datos específica. Es la encargada en última instancia de *pasivar* y *activar* objetos en el almacenamiento persistente. Esta clase implementa la interfaz `dbObjectAdaptor` del paquete `dbcollection`.

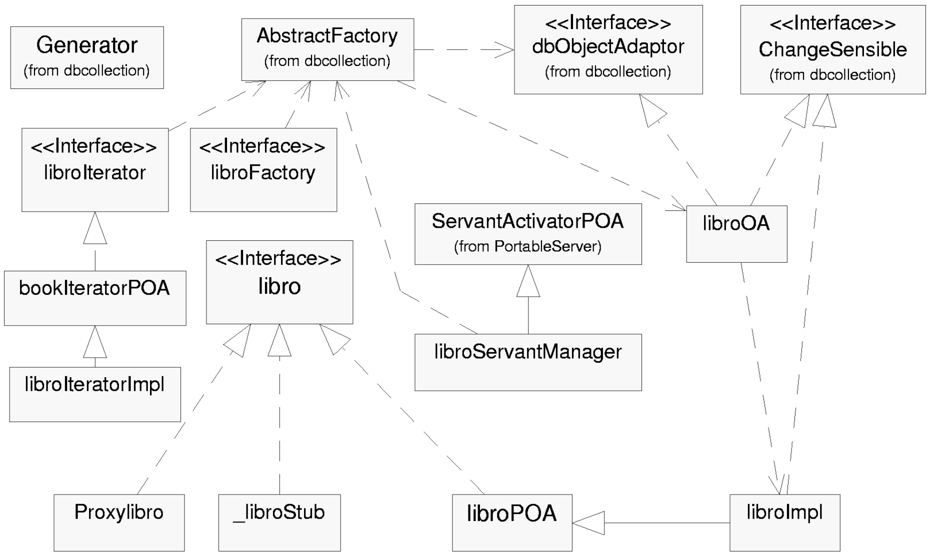

Figura 3. Diagrama UML (parcial) de las clases del *framework* de persistencia.

En la Figura 3 se muestra la relación existente entre estas clases. En la parte superior se muestran las clases e interfaces pertenecientes al *framework*. Las clases creadas por el Generador se muestran debajo (junto con otras proporcionadas por el ORB). A la izquierda se muestran las clases que hacen de interfaz hacia los clientes, y a la derecha las que implementan estos interfaces.

Desde el punto de vista de la implementación, la factoría representa, además, un lugar donde implementar uno de los requisitos impuestos a la implementación: la política de *caché* que se seguirá con los objetos: como administrador del conjunto de objetos, decidirá cuándo *pasivar* o *activar* cada objeto, esto es, cuándo guardar el estado de un objeto en la base de datos y cuándo traer un objeto de la base de datos a memoria.

Cada factoría delega el manejo de su *caché* en la clase `AbstractFactory` del paquete `dbcollection`. Esta factoría abstracta implementa la política de *caché* que se seguirá con los objetos. `AbstractFactory` maneja los objetos CORBA como entidades de la clase `java.lang.Object` (nótese que todo objeto CORBA implementado en Java hereda, finalmente, de esta clase). La *caché* se implementa con un *array* asociativo (`Hash`) que liga la clave de cada objeto (implementada como un `Vector` de `java.lang.Object`) con el objeto–implementación–en sí. La clase `AbstractFactory` incluye los siguientes métodos:

```
// Java
package dbcollection;
class AbstractFactory {
    public java.lang.Object getObject(Vector key) {...}
    public void store(Vector key) {...}
    public void remove(Vector key) {...}
    public boolean exists(Vector key) {...}
    // ...
}
```

En el manejo de la *caché*, la factoría abstracta debe finalmente acceder a la base de datos a recuperar/guardar un objeto determinado. Debido a que `AbstractFactory` maneja objetos tratándolos como pertenecientes a su clase menos específica, necesita delegar esta funcionalidad a clases especializadas. Para cada tipo de objetos persistente se genera una clase "adaptadora"[3] que implementa la interfaz `dbObjectAdaptor` del paquete `dbcollection`. Estas clases adaptadoras poseen una interfaz muy sencilla, que permite a la factoría recuperar un objeto de la base de datos, almacenar un objeto en la base de datos, preguntar por la existencia de un objeto en la base de datos, etc. Cada clase adaptadora es capaz de "serializar" a un tipo de objetos en la base de datos y de recuperar a un objeto dado de la base de datos, conociendo su clave.

Referencias CORBA Persistentes El modelo de objetos CORBA especifica que, para que un cliente pueda obtener los servicios de un objeto servidor, el primero debe obtener una *referencia* al segundo. A partir de la obtención de la referencia, el cliente puede realizar invocaciones (peticiones de servicios) sobre el objeto servidor. Para que un objeto sirva peticiones, debe estar localizado en memoria. Sin embargo, debido a que el conjunto de objetos servidores presentes en la base de datos puede llegar a ser potencialmente grande, es posible que algunos de esos objetos, por la política de caché, no estén en memoria en el momento de la petición (hayan sido *pasivados*). Así, se requiere un esquema de ctivación bajo demanda Este esquema resulta muy problemático con el BOA, aunque perfectamente posible con el POA (*Portable Object Adapter*) tras seleccionar las políticas adecuadas [9, 10].

Referencias Persistentes: caso BOA Básicamente, el BOA se puede ver como un intermediario entre clientes y servidores que es capaz de (1) mantener una

[3] No se deben confundir estas clases "adaptador" con el adaptador de objetos CORBA.

istade objetos servidores activos, y (2) redirigir la petición de un cliente al objeto servidor correcto.

Aunque, como se reconoce en la especificación del POA [1], el rango y tipo de servidores presentes en una aplicación puede ser elevado (lo cual requiere de la definición y el establecimiento de un conjunto de políticas) la pobre funcionalidad del BOA, a menudo se ha sustituido simplemente por un par de operaciones del ORB: `connect` y `disconnect`.[4]

El problema crucial que aparece con el BOA es que el ciclo de vida de los objetos implementación está ligado al de las referencias a los mismos, es decir, una referencia a un objeto CORBA no es válida si el objeto implementación correspondiente no está activo en la parte servidor. Mantener activos y en memoria a todos los objetos implementación que representan a entidades de la base de datos es irrealizable. El verdadero dilema es la "persistencia" de las referencias a objetos generadas. Mientras que los clientes de los objetos persistentes esperan que las referencias que se obtuvieron inicialmente de la factoría sean válidas durante el tiempo que las necesiten, en realidad, los objetos que dan implementación a esos tipos persistentes deben, en algún momento, y guiados por la política de caché, abandonar la memoria principal y desactivarse en el ORB del servidor.

La cuestión resulta algo paradójica: ¿cuánto puede esperar un cliente que persista su referencia a un objeto persistente? En el caso del BOA, no se puede dar una respuesta fiable a esta pregunta.

Nuestra solución fue que el Generador construyera unas clases *wrapper* (siguiendo el patrón *Proxy* [12]) que amortiguaran el caso en el que una referencia hubiera dejado de ser válida debido a que el objeto servidor correspondiente se hubo de *pasivar* por el sistema de *caché*. Así, para cada interfaz (por ejemplo `libro`), se genera una clase `Proxy`*`libro`*, con los mismos métodos *accessor/mutator* que el objeto persistente, pero aprovechando que CORBA retorna una excepción `CORBA::OBJECT_NOT_EXIST` en el caso de que el objeto servidor no exista o no esté activado. El inconveniente de estas clases es que el manejo de errores se hace en la parte cliente. La clase "segura" necesitará:

1. La referencia (`ref`) más reciente al objeto persistente del que se encarga,
2. La clave del objeto (`key`) del que se encarga, para poder recuperarlo en el caso de que la referencia actual se vuelva inválida.
3. Una referencia a la factoría para ese tipo de objetos (`factory`), para recoger la nueva referencia al objeto en caso de que haya dejado de ser válida.

Así, para cada método (*accessor*, en este caso) de la interfaz original, el correspondiente de la clase "proxy" será, esquemáticamente:

[4] Así ocurre, por ejemplo, en el *mapping* estándar de CORBA a Java [16], en el que no se incluyó el BOA, si bien es cierto que la mayoría de empresas proveedoras lo implementó con mayor o menor suerte.

```
class Proxylibro {
    public tipo AtributoX() {
        while (true) {
            try {  return ref.AtributoX();  }
            catch(CORBA.OBJECT_NOT_EXIST) {
                ref = factory.getObject(key);
            }
        }}}
```

Referencias Persistentes: caso POA El caso POA es bastante diferente. En la especificación del POA (parte del estándar CORBA [1]) se hace la distinción entre las *referencias* y los *objetos servidores* (llamados *servants* en la especificación) encargados de darles implementación. Así, se separa el ciclo de vida de los *servants* de los de las referencias, pudiendo crear referencias persistentes cuyo tiempo de vida supera al del *servant* encargado de implementarlo e incluso del proceso servidor en el que éste último está ejecutándose. Incluye siete políticas que se pueden configurar en concordancia con las necesidades de los objetos servidor. De entre todas las políticas, estamos interesados en: (1) Generar referencias persistentes; (2) Asignar a cada objeto una clave que corresponda con la clave de la Base de Datos; y (3) Tomar el control en el momento en el que un cliente requiera un servicio de un objeto implementación que no esté "activo" (activación bajo demanda).

El primer punto se consigue especificando el valor `PERSISTENT` en la política `Lifespan` (tiempo de vida). El segundo, con el valor `USER_ID` en la política `IdAssignment` (asignación de identificador), y el tercero, con el valor `USE_SERVANT_MANAGER` de `RequestProcessing` (procesado de peticiones).

No establecemos el valor `NON_RETAIN` en la política `ServantRetention` (registro de *servants*), sino que utilizamos el valor por defecto `RETAIN`, lo cual obliga al POA a mantener un registro de los *servants* activos en su *Active Object Map*. Aunque el valor `NON_RETAIN` de esta política se recomienda en los casos en los que el POA tiene que servir a un conjunto potencialmente grande de objetos (como es este caso), utilizamos el valor `RETAIN` por dos razones: (1) el proceso de elección del *servant* se hace mucho más rápido, ya que primero se consulta en el *Active Object Map* y después al *ServantManager* sólo en el caso en el que el objeto no se encuentre activo; (2) el conjunto de objetos que permanece activo en un momento dado está limitado por el tamaño de la *caché*: el manejo de la misma realiza la activación y desactivación de objetos servidores de forma explícita utilizando los métodos del POA `activate_object_with_id` y `deactivate_object`. La elección de este valor junto con `USE_SERVANT_MANAGER` implica también el uso de un *ServantActivator*.

Es claro que en el caso POA las clase "seguras" no son necesarias. Sin embargo, el Generador debe crear un *ServantManager* (en concreto un *ServantActivator*) para cada interfaz. Así, para la interfaz `libro`, se generará la clase correspondiente *`libro`*`ServantManager`. Esta clase posee dos métodos: `incarnate`, encargada de activar objetos bajo demanda, y `etherealize`, vacío en nuestra implementación ya que la desactivación la realiza el mecanismo de *caché*. Por ejemplo, para la interfaz `libro`, el `ServantActivator` sería el siguiente:

```
public class libroServantManager
    extends org.omg.PortableServer.ServantActivatorPOA {
    private AbstractFactory af;
    public libroServantManager(AbstractFactory af) { this.af = af; }
    public org.omg.PortableServer.Servant incarnate(
        byte[] oid, org.omg.PortableServer.POA adapter)
        throws org.omg.PortableServer.ForwardRequest {
        try {
            ObjectInputStream ent =
                new ObjectInputStream(new ByteArrayInputStream(oid));
            Vector key = (Vector)ent.readObject();
            return (org.omg.PortableServer.Servant)af.incarnateObject(key);
        } catch (Exception ex) { ex.printStackTrace(); }
        throw new org.omg.CORBA.OBJECT_NOT_EXIST();
    }
    public void etherealize(byte[] oid, ... ) {
        // Vacío
    }
}
```

A la clase `AbstractFactory` se le debe dotar de un método `incarnateObject` que ofrezca la posibilidad al `ServantActivator` de indicarle qué objeto se acaba de activar para que el `AbstractFactory` lo incluya en el manejo de caché. La diferenciación entre los métodos `getObject` e `incarnateObject` responde a la distinta semántica asignada a los mismos. Mientras que el primero se utiliza cuando un cliente requiere de forma *explícita* un nuevo objeto dada su clave, el segundo es llamado de forma *implícita* cuando un cliente quiere utilizar un objeto que ha sido desactivado por el manejo de caché. Esta diferencia semántica se traduce en que mientras que `getObject` debe, en algunos casos, registrar el nuevo objeto en el POA, `incarnateObject` no tiene que registrarlo nunca, ya que es llamada como resultado de una activación bajo demanda por el POA.

4 Conclusiones y Trabajo Futuro

En este trabajo hemos presentado un *framework* que permite generar un soporte de persistencia de forma automática para objetos servidores CORBA. Las aproximaciones tradicionales hacia la persistencia de objetos CORBA han adolecido de dos inconvenientes: incompatibilidades (en el caso de soluciones ofrecidas por los productos comerciales) y de diseños poco flexibles en el caso de soluciones *ad hoc*, ambos motivados por la deficiencia en la especificación del ciclo de vida para objetos servidores CORBA.

También destacamos las siguientes características del *framework* presentado: i) Persistencia transparente con respecto a los clientes; ii) *Wrapping* automático de bases de datos heredadas; iii) Integración automática de los objetos persistentes con el resto de definiciones IDL de la aplicación; iv) Es una solución basada sólo en estándares y por lo tanto portable; v) Facilidad para integrar distintos modos de almacenamiento.

Para completar el *framework* es necesario implementar los siguientes aspectos: i) posibilidad de especificar agregación y atributos multivaluados en la declaración de una interfaz DDL; ii) transacciones; iii) evaluar los servicios ofrecidos por los nuevos estándares CORBA [1], como el estándar de paso de objetos por valor (*Objects By Value*, OBV), así como distintas configuraciones de las políticas del POA; iv) caché en los clientes, lo que obligaría a implementar mecanismos de coherencia de caché; v) estrategias de replicación, tolerancia a fallos y notificación de eventos a través de servicios de nombres o eventos. También planeamos comparar la eficiencia con una implementación C++.

5 Bibliografía

1. *Common Object Request Broker Architecture, revision 2.3.1*. OMG document orbos/99-10-07. Object Management Group, 1999.
2. *Objetos distribuidos y persistencia con CORBA, Java y C++*. Javier Carlos Ros Sánchez. Proyecto Fin de Carrera, Facultad de Informática, Universidad de Murcia, 1999.
3. *Object Management Group*. `http://www.omg.org`.
4. *CORBAServices*. OMG document orbos/98-12-09, Object Management Group, 1998.
5. *What We Are Missing in the CORBA Persistent Object Service Specification*. Kleindienst, J., Plášil, F., Tuma, P. OOPSLA'96 Workshop on Large Persistent and Distributed Systems, 1996.
6. *Orbix Programmer Guide*. IONA Technologies, 1998. `http://www.iona.com`.
7. *Visibroker Programmer Guide*. Inprise, 1999. `http://www.inprise.com`.
8. *Persistent State Service 2.0 RFP*. OMG document orbos/97-06-07, Object Management Group, 1997.
9. *Advanced CORBA Programming with C++*. Henning, M., Vinoski, S. Addison-Wesley, 1999.
10. *Using the Portable Object Adapter for Transient and Persistent CORBA Objects*. Schmidt, D. C., Vinoski, S. SIGS C++ Report. Vol. 10, No. 4, April 1998.
11. *Persistent State Service 2.0 Joint Revisted Submission*. OMG document orbos/99-07-07. Fujitsu et al., 1999.
12. *Design Patterns: Elements of Reusable Software*. Gamma, E., Helm, J., Johnson, R., Vlissides, J., Addison-Wesley, 1995.
13. *The Java Language Specification*. Gosling, J., Joy, B., Steele, G. Addison-Wesley, 1996.
14. *JDBC API Tutorial and Reference*.White, S., Fisher, M., et al. Addison-Wesley, 1999.
15. *A guide to the SQL Standard*. Date C. J., Darwen, H. Addison-Wesley, 3ed. 1993.
16. *IDL-Java Language Mapping Specification*. OMG document orbos/99-07-53. Object Management Group, 1999.
17. *The Design of a Robust Persistence Layer for Relational Databases*, Scott W. Ambler. AmbySoft Inc. White Paper, 1999.