Diseño de un Sistema de Persistencia Implícita Mediante Reflectividad Computacional

Ortín Soler, F. Martínez Prieto, A.B. Álvarez Gutiérrez, D., y Cueva Lovelle, J.M.

Departamento de Informática, Universidad de Oviedo
Calvo Sotelo 33007, Oviedo, España
{ortin, belen, darioa, cueva}@pinon.ccu.uniovi.es

Abstract. Actualmente la utilización de SGBDOOs o sistemas de persistencia carecen de flexibilidad por la necesidad de incluir código adicional al propio de la aplicación y por la necesidad de amoldarse a las distintas APIs y versiones existentes.
En este artículo se plantean las posibilidades de un sistema integral orientado a objetos, basado en una máquina abstracta dotada de reflectividad. Gracias a ésta se puede diseñar un sistema de persistencia implícita sin necesidad de modificar el código de una aplicación. El sistema de persistencia se apoyará en el motor de la base de datos propio del sistema integral.
Se define la característica de reflectividad computacional como inherente al sistema. Basándonos en esta propiedad, tanto el diseño del motor de bases de datos como la programación de aplicaciones poseerán una gran flexibilidad. La modificación dinámica de los distintos parámetros del sistema podrá ser llevada a cabo en función de los requisitos de la aplicación
La reflectividad permite el diseño de un *middleware* capaz de conseguir un sistema de persistencia implícita para el programador, permitiendo la selección transparente y en tiempo de ejecución de un nivel de persistencia para los objetos de la aplicación, sin necesidad de incluir código adicional. Con ello se consigue una programación orientada a objetos de muy alto nivel, totalmente portable y flexible en tiempo de ejecución.

1 Introducción. SGBDOOs y Lenguajes Persistentes

1.1 SGBD Orientados a Objetos

Los SGBDOO deben su aparición principalmente al surgimiento de nuevos tipos de aplicaciones para las que el modelo de objetos representaba mejor la semántica de la nueva información a almacenar. Todos estos sistemas se caracterizan por emplear el modelo de datos orientado a objetos y la integración con los lenguajes de programación existentes [1]. Ejemplos ya clásicos de estos sistemas son entre otros muchos ObjectStore, O2, GemStone, POET y VERSANT. La diferencia entre unos sistemas y otros venía impuesta principalmente por la elección del lenguaje de programación y el lenguaje de consulta, y era esta diferencia precisamente la que limi-

taba la portabilidad entre unos sistemas y otros. Con el fin de reducir estas limitaciones surge un intento de estandarización [2, 3] propuesto por el Object Database Management Group (ODMG 1.0 y ODMG 2.0). Actualmente la mayoría de estos sistemas ya clásicos se proclaman como 'compliant' con lo propuesto por ODMG.

Paralelamente a estos sistemas y ante las nuevas necesidades, se produce también una evolución de los sistemas relacionales constituyendo los sistemas de bases de datos relacionales extendidos. Dichos sistemas se caracterizan por combinar la orientación a objetos con el modelo relacional realizando los cambios necesarios tanto a nivel de gestor de almacenamiento como de datos del SGBDR. Sistemas ya clásicos son por ejemplo UniSQL y Persistence.

1.2 Lenguajes de Programación Persistentes

Actualmente, coexistiendo con estos sistemas anteriormente mencionados están surgiendo nuevas iniciativas, la mayoría de ellas ideadas para conseguir que los lenguajes de programación soporten objetos persistentes.

Si nos centramos en el lenguaje de programación Java, por ejemplo, vemos que existe un amplio abanico de posibilidades cara a conseguir dicho objetivo. Así, nos encontramos con proyectos como *PJava* (Persistent Java) [4], que proporciona un entorno de programación persistente para el lenguaje Java basado en una modificación de su plataforma.

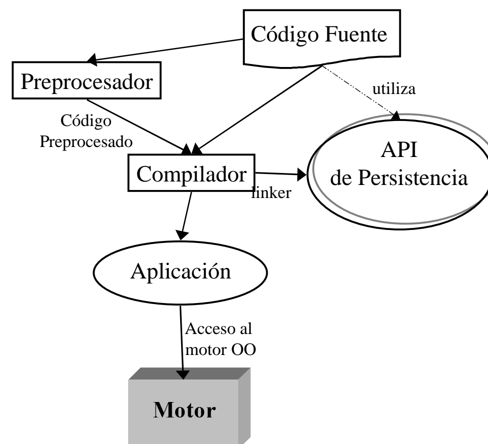

Fig. 1. Esquema general para la inclusión de funcionalidades de bases de datos en un lenguaje de programación.

Otras iniciativas de cara a conseguir la persistencia de objetos Java se basan en emplear motores de almacenamiento, tales como PSE y PSE Pro [5] para Java (también lo hay para C++), que permitan almacenar y recuperar objetos en su formato nativo. Estos motores persistentes proporcionan generalmente APIs que dotan al lenguaje de las funcionalidades de bases de datos. Tal es el caso de Jeevan [6], una base de datos para un solo usuario sobre la plataforma de Java que proporciona un API

sencillo formado por cuatro clases y dos interfaces. Éste permite la especificación de índices y las consultas dinámicas.

Si lo que se pretende es traducir los objetos Java a un modelo relacional existen también multitud de opciones. Desde bases de datos en las que es el programador el responsable de la traducción a tablas (StreamStore[7]), a otros productos como los Java Blend [8], en los que los objetos son traducidos a tablas (o viceversa) de una forma transparente para el usuario.

1.3 Añadiendo Funcionalidades de Bases de Datos a un Lenguaje de Programación

La inclusión de las funcionalidades de bases de datos en un lenguaje de programación suele venir generalmente representado mediante una *Application Programming Interface* (API), como muestra el esquema de la figura 1, y empleando en algunos casos un preprocesador.

Los inconvenientes de este esquema de trabajo son:

1. Complejidad para el usuario. El usuario tiene que conocer las ampliaciones de los propios lenguajes o la utilización de unas APIs.
2. La legibilidad y mantenimiento del código se ven afectadas por la existencia de un código adicional (código intruso) que se debe añadir al propio de la aplicación para conseguir la persistencia de los objetos.
3. Se pierde portabilidad en el código por la dependencia entre el API utilizado y la aplicación.
4. Flexibilidad restringida. El cambio en el modo en que se accede al motor de la base de datos, implica modificar el código y recompilar la aplicación de nuevo.

1.4 Persistencia Implícita en un Sistema Integral Orientado a Objetos Reflectivo

En este artículo proponemos la persistencia implícita como una nueva forma de añadir la funcionalidad de persistencia a un lenguaje de programación. El usuario no tiene que codificar cómo se gestiona la persistencia de los objetos con código intruso. Se eliminan los problemas de complejidad, legibilidad y flexibilidad mencionados. El sistema definirá un subsistema de persistencia que se apoyará en el motor de la base datos.

En la mejora de la flexibilidad, el subsistema de persistencia será capaz de llevar a cabo cambios dinámicos en los servicios pedidos al motor (como por ejemplo un cambio en el tipo de índice utilizado) sin necesidad de modificar la aplicación. Esta investigación esta siendo llevada a cabo sobre el sistema integral orientado a objetos Oviedo3, basado en una máquina abstracta reflectiva. Tanto el motor como el subsistema de persistencia han sido incorporados al sistema integral, para su utilización en el desarrollo de aplicaciones.

El resto del documento está organizado de la siguiente manera: La sección 2 justifica el desarrollo de un sistema integral orientado a objetos, describiendo brevemente sus características y la máquina abstracta. Además se describen las ventajas de integrar el motor de la base de datos dentro del sistema. En el apartado 3 se introduce el concepto de reflectividad y cómo es utilizado en el diseño de la máquina abstracta. En

la sección siguiente se identifican el sistema de persistencia y el motor de la base de datos coexistiendo en el sistema integral para llegar a alcanzar lo que definiremos como persistencia implícita. Las ventajas del sistema propuesto serán expuestas en la sección 5. En la sección 6 comentaremos trabajos relacionados y finalmente las conclusiones se realizarán en la sección 7.

2 Sistema Integral Orientado a Objetos

La utilización del paradigma de la orientación a objetos no es normalmente llevada a cabo en todos los componentes de un sistema. Existen por ejemplo, lenguajes de programación, bases de datos e interfaces de usuario basados en la orientación a objetos, pero que tienen que adaptarse a otro paradigma distinto al interactuar con otro elemento del sistema como por ejemplo el operativo. Esto se traduce en un problema de adaptación de paradigmas en el que es necesaria una traducción entre uno y otro. En ocasiones se implementan estas traducciones con un código adicional. Un ejemplo típico es el almacenamiento de objetos en una base de datos relacional.

Nuestra proposición para resolver estos problemas pasa por el desarrollo de un sistema en el que la orientación a objetos sea la característica común a todos sus módulos. Oviedo3 [9] es un proyecto de investigación en el que se está desarrollando un sistema integral orientado a objetos. Todos sus componentes –aplicaciones, interfaces de usuario, lenguajes de programación, compiladores, bases de datos y el propio sistema operativo –comparten el paradigma de la orientación a objetos.

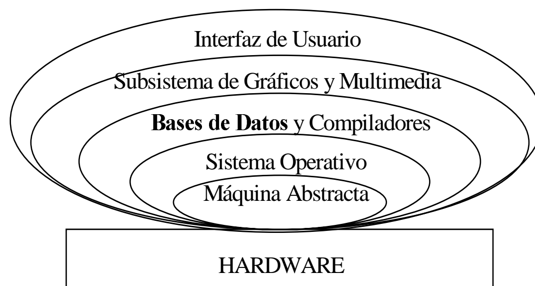

Fig. 2. Componentes del sistema Oviedo3.

El sistema utiliza una abstracción básica: el objeto. Se pueden crear objetos, destruirlos y mandar mensajes a éstos. Los objetos se ejecutan sobre la especificación de una máquina abstracta orientada a objetos. Esta máquina ofrece el modelo básico de objetos y es el soporte para todos los objetos del sistema. Su característica de reflectividad le dota de un alto nivel de flexibilidad.

El sistema operativo está formado por un conjunto de objetos –del mismo tipo que el resto de objetos del sistema –que proporcionan la funcionalidad propia del operativo. Así pues, nuestro sistema consiste en un entorno de objetos en el que su funcionamiento viene dado por el modo en el que éstos intercambian mensajes, sin distinción de a qué subsistema pertenece cada uno (figura 3). Actualmente CORBA es un ejemplo de búsqueda de un entorno heterogéneo de objetos distribuidos.

2.1 La Máquina Abstracta y el Modelo de Objetos

La máquina abstracta proporciona el soporte básico para el modelo de objetos del sistema. El objetivo es aprovecharse de los conceptos existentes en la orientación a objetos así como de su semántica. Las características del modelo incluyen:

- Identidad única de objetos (accesibles por referencias);
- Encapsulamiento (acceso al objeto a través de sus métodos);
- Clases (utilizadas también para derivar tipos);
- Relación de herencia (es-un) múltiple;
- Relación de agregación (es-parte-de);
- Relación de asociación (relacionado-con);
- Polimorfismo y comprobación de tipos;
- Manejo de excepciones;
- Concurrencia, distribución y persistencia.

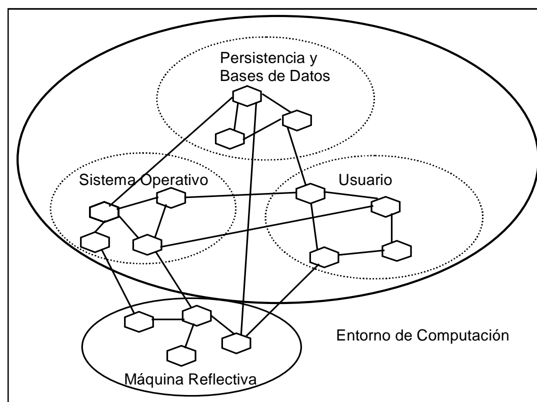

Fig. 3. Entorno de computación compuesto por un conjunto de objetos homogéneos.

El lenguaje de la máquina abstracta es un lenguaje de bajo nivel OO. Permite la declaración de clases, métodos y manejo de excepciones. Existen objetos básicos proporcionados por la máquina y sobre éstos se pueden crear otros nuevos.

2.2 Motor de BDOO en el Sistema Integral

El motor de BDOO es parte del entorno computacional del sistema integral. Los objetos gestores de la base de datos son objetos como los del sistema operativo o como los de una aplicación de usuario, que proporcionan la funcionalidad de la base de datos: persistencia, indexación, procesamiento de consultas, etc. Todos se agrupan en un subsistema.

La existencia de un sistema operativo y de una máquina abstracta orientada a objetos, produce una serie de beneficios a la hora de construir el motor [10]:

- Integración total con el resto del sistema. El SGBDOO puede ser visto como el gestor de ficheros en un sistema operativo convencional. Éste se encarga de ges-

tionar los objetos persistentes. Las bases de datos no se utilizarán de forma aislada. Su uso vendrá desde un lenguaje de programación hasta la línea de comandos del sistema operativo.

- Facilidad en su implantación. El motor se apoyará por un lado en la reflectividad estructural de la máquina, y por otro lado en los servicios del sistema operativo (seguridad, concurrencia, etc.). Su diseño se hará de forma incremental reutilizando los objetos ya existentes.
- Mejora en ejecución. No es necesario añadir nuevas capas de software para cubrir el salto semántico que existe entre los distintos paradigmas utilizados en muchos sistemas actuales.
- Mejora en la productividad. La programación de aplicaciones que utilicen bases de datos es más productiva. No es necesario cambiar el paradigma de programación y la utilización de los servicios de la base de datos es uniforme en todo el sistema.

El motor es flexible y fácilmente configurable, apoyándose en la extensibilidad proporcionada por la orientación a objetos. De todos los elementos del motor, centraremos el punto de atención en el mecanismo de indexación en el que se basará el procesamiento de consultas, y sobre el que se está trabajando actualmente.

2.3 Técnicas de Indexación

La existencia de jerarquías de herencia y de agregación, así como la posible invocación de métodos en los lenguajes de consulta orientados a objetos exigen la existencia de mecanismos de indexación que permitan un procesamiento eficiente de las consultas bajo estas condiciones. Son muchas las técnicas de indexación en orientación a objetos que se han propuesto y que según [11] se pueden clasificar en:

1. Estructurales. Se basan en los atributos de los objetos. Estas a su vez pueden clasificarse en técnicas que proporcionan soporte para consultas basadas en la *Jerarquía de Herencia* (por ejemplo, SC [12], CH-Tree [12], H-Tree [13], etc.), técnicas que proporcionan soporte para la *Jerarquía de Agregación* (ejemplos son entre otros [14] Nested, Path y Multiindex), y técnicas que soportan tanto la *Jerarquía de Agregación como la Jerarquía de Herencia* (Ejemplo Nested Inherited [11])
2. De Comportamiento. Proporcionan una ejecución eficiente para consultas que contienen invocación de métodos. La materialización de métodos [15] es una de dichas técnicas.

2.4 Mecanismo de Indexación de Oviedo3

En función del tipo de consulta que sea más frecuente sobre una determinada clase (o jerarquía de clases) puede ser más eficiente el empleo de una de esas técnicas que otras. Sin embargo, en la mayoría de los SGBDOO existentes en el mercado el mecanismo de indexación es uno o un conjunto fijo de ellos, y el diseñador no tiene posibilidad de seleccionar el esquema de indexación, ni por supuesto añadir nuevos esquemas.

El motor aquí mostrado presentará las siguientes características:

- Distintos esquemas de indexación. El sistema permitirá el uso de distintos esquemas de indexación. Inicialmente SC, CH-Trees y Path Index se han introdu-

cido en un primer prototipo implementado. Sin embargo, el mecanismo de indexación permite fácilmente incluir nuevos esquemas (gracias al polimorfismo).

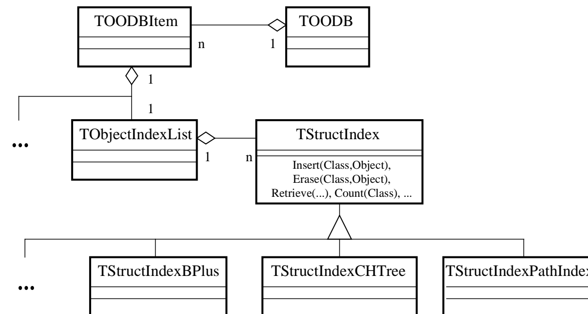

Fig. 4. Diagrama de clases del mecanismo de indexación.

- Selección de mecanismo de indexación. Permitirá seleccionar el mecanismo de indexación que se considere más apropiado en función del tipo de consulta de que va a ser objeto la clase (o jerarquía de clases) que se está tratando. Gracias a la herencia y polimorfismo esta selección puede llevarse a cabo fácilmente (con las clases *TObjectIndexList* y *TStructIndex*).
- Independencia del tipo de dato. Implica que el índice puede ser ejecutado sobre cualquier tipo de datos (no simplemente sobre los tipos simples). Para conseguir esto, el mecanismo permite al usuario definir sus propios operadores de comparación.

El siguiente código es un ejemplo de el método *NewIndex* de la clase *TObjectIndexList* (figura 4):

```
/ * Distintos Índices de una Clase
Persistent CLASS TobjectIndexList

AGGREGATION
  / * Lista de Índices
  Persistent IndexList:TList;
  / * Índice Seleccionado
  Persistent Selected:Integer;

METHODS
  / ...

/ * Añade un nuevo Índice
NewIndex(anIndex:TStructIndex):Bool
  REFS
    bRes:Bool;
    Number:Integer;
  INSTANCES
    One:Integer(1);
    Zero:Integer(0);
CODE

    / * Añade el índice
    IndexList.Add(anIndex);
    / * Number=size(IndexList)
    IndexList.GetSize():Number;
    / * ¿Es number igual a 0?
    Number.Equal(One):bRes;
    Delete Number;
    JFD bRes,End; / Jump if false

   / * Se selecciona si es el 1º
    Selected.Set(Cero);

  End:
    Exit;
  ENDCODE

ENDCLASS
```

3 Reflectividad en el Sistema Integral Orientado a Objetos

La reflectividad es la propiedad de un sistema computacional que le permite razonar y actuar sobre él mismo así como ajustar su comportamiento en función de ciertas condiciones [16]. El dominio computacional de un sistema reflectivo añade al dominio de un sistema convencional, la estructura y la computación de sí mismo. Podremos definir así dos tipos de reflectividad [17]:

- Reflectividad estructural: es la más obvia y actualmente la más conocida. Se refiere al estado estructural del sistema en tiempo de ejecución [18]. El *Java Reflection API* [19] es un ejemplo de reflectividad estructural.
- Reflectividad computacional: es la posibilidad que tiene un proceso para describirse, analizarse y modificarse en tiempo de ejecución.

La portabilidad de nuestro sistema integral viene dada por el código binario de una máquina abstracta que garantiza una plataforma independiente [20]. Para poder definir el Sistema Integral sobre esta máquina, debemos llegar a un compromiso crítico en la definición ésta. En un primer prototipo se desarrolló una máquina abstracta orientada a objetos con posibilidades de herencia, polimorfismo, manejo de excepciones y multitarea [21]. Posteriormente se fueron añadiendo propiedades como el sistema de persistencia [22], distribución y seguridad [23].

Una posibilidad para aumentar la funcionalidad de la máquina es ir añadiendo nuevas instrucciones al repertorio así como la interpretación correspondiente que las define, implementando así la nueva funcionalidad deseada. Este no es un mecanismo de ampliación genérico puesto que cada nueva ampliación conlleva una nueva versión y recompilación de la máquina (emulador). Por otro lado, el repertorio de la máquina constituye la interfaz con los programas, y no es conveniente que cambie porque eso obligaría a adaptar los programas ya escritos para aprovechar la nueva funcionalidad.

Otra posibilidad más flexible se basa en introducir el concepto de reflectividad estructural en la máquina. Se define como unidad de computación el objeto y sobre éste una serie de primitivas (invocación de métodos fundamentalmente). Los objetos poseen reflectividad estructural de forma que se podrá conocer en todo momento su estructura y modificar todas sus propiedades excepto las definidas como primitivas.

Con este diseño se implementará cualquier propiedad que se desee introducir de manera dinámica y que pueda ser expresada mediante las primitivas. En lugar de modificar el simulador de la máquina, la funcionalidad adicional vendrá codificada en código binario de la máquina que se apoyará en la reflectividad estructural de ésta. Esta línea de diseño de máquinas abstractas también es adoptada por Smalltalk-80 [24] y sobre todo por ObjVLisp [25].

3.1 Reflectividad Computacional

La verdadera potencia del sistema que definimos en este artículo viene dada por la reflectividad computacional. Para obtener un sistema con esta propiedad tenemos que apoyarnos en la torre de intérpretes propuesta por Smith [26]. En concreto definimos dos niveles de computación en esta torre:

- La ejecución de un intérprete de un lenguaje *L*, en código binario de la máquina virtual *B*.

- La ejecución de un programa de usuario en el lenguaje *L*, por el intérprete.

Para conseguir que el sistema expresado en el lenguaje *L* sea computacionalmente reflectivo identificamos las dos siguientes necesidades:

1. Que se pueda "saltar"[1] de la computación de *L* a la de *B* en la torre de dos niveles.
2. Que la computación de *B* posea reflectividad estructural para poder modificar el estado de computación de *L*.

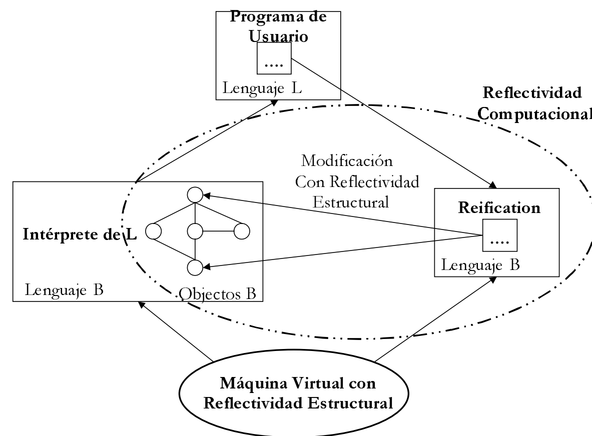

Fig. 5. Reflectividad computacional sobre una máquina abstracta con reflectividad estructural.

Actualmente existen varios sistemas de reflectividad como los MOPs [27]. Un MOP (Meta Object Protocol) es la interfaz que presentan unos objetos auxiliares con la posibilidad de modificar los objetos ya existentes. Esta interfaz es definida en tiempo de compilación, lo que permite una mayor eficiencia a costa de una pérdida de flexibilidad. Otros como MetaJava son más flexibles, al identificar este protocolo en tiempo de ejecución, y algo menos eficientes [28]. Un grado mayor de flexibilidad[2] es dado por el salto en los niveles de computación propuesto.

En el resto de este artículo identificaremos la reflectividad computacional como simplemente reflectividad.

4 Obtención de Persistencia mediante Reflectividad

Como se ha comentado, el sistema de persistencia viene dado por un motor de bases de datos orientadas a objeto. Para que éste sea lo más flexible posible, tan sólo se define su interfaz dejando sin especificar su implementación como se hace en otras arquitecturas como CORBA [29]. Este diseño posibilita además lo que se conoce como *implementaciones abiertas* [30]: la existencia de varias implementaciones que

---

[1] Este tipo de salto se denomina reification [Maes87]

[2] La flexibilidad total se consigue a costa de perder en eficiencia. En el sistema que proponemos dejamos la eficiencia como futuras optimizaciones anteponiendo la flexibilidad a ésta.

puedan ser seleccionadas en función de una serie de parámetros, como por ejemplo, los mecanismos de indexación antes mencionados.

Otra de las características propias de este tipo de programación es la portabilidad. Al igual que en la plataforma Java [31], se consigue portabilidad del código binario de esta máquina abstracta en varios sistemas. Añadido a la portabilidad del código de la aplicación, debemos estudiar los accesos a otros objetos externos a ésta. Aquí es donde se encuentran los objetos del sistema operativo y en concreto el motor de la base de datos. En el caso de que migremos la aplicación a otra máquina que soporte el sistema integral, puede darse que el nuevo motor tenga una implementación distinta aprovechando determinadas características del nuevo sistema físico. En este caso la portabilidad también está asegurada gracias a la especificación común de la interfaz del motor.

La implementación del motor se podrá apoyar en la reflectividad estructural proporcionada por la máquina abstracta. Esto hace que se simplifique mucho la programación del motor puesto que se podrá acceder a todas las propiedades de los objetos en tiempo de ejecución. De esta forma la creación, consulta, modificación y destrucción de los objetos y de sus propiedades son servicios proporcionados por la propia máquina gracias a su condición de reflectividad.

Una vez especificada la interfaz del motor y habiendo realizado la implementación de sus funcionalidades, sólo nos queda utilizarlo como sistema de persistencia concreto en la programación de aplicaciones.

4.1 Aplicaciones con Persistencia de Objetos Explícita

En las aplicaciones que se desee utilizar la persistencia del sistema integral de forma explícita se deberá limitar a utilizar los servicios del motor de la base de datos expuestos mediante su interfaz. Para acceder al motor, se obtendrá una referencia a éste puesto que será un objeto propio del Sistema Integral [9].

En esta *persistencia explícita* es el programador el que identifica en qué momento se utiliza un determinado servicio del motor. Este tipo de programación es más eficiente que la *persistencia implícita* (siguiente punto) pero sigue siendo el programador el que asume toda la responsabilidad de gestionar los objetos persistentes de una forma correcta. Dado que el motor se encuentra en el ámbito del Sistema Operativo y es accesible desde cualquier parte del sistema[3], el acceso a las bases de datos se produce desde el propio lenguaje de programación, como es el caso por ejemplo de PSE Pro, Jeevan, etc.

4.2 Aplicaciones con Persistencia de Objetos Implícita

En la persistencia implícita la aplicación no tiene que realizar ninguna acción especial para hacer persistir los objetos. El sistema hace persistir los objetos de manera transparente, no existiendo código adicional al propio de la lógica de la aplicación. Es decir, no hay necesidad de especificar las distintas llamadas al motor de la base de

---

[3] Cabe destacar que en el Sistema Integral Orientado a Objetos la máquina abstracta trabaja con referencias distribuidas, por lo que se podrá acceder a Sistemas de Persistencia remotos.

datos [22]. La responsabilidad de enlazar la aplicación con el sistema de persistencia ya no será tarea del programador. Con este tipo de persistencia, se puede crear una aplicación, depurarla, ejecutarla y posteriormente (en tiempo de ejecución) identificar ciertos objetos como persistentes incluso con diferentes niveles de persistencia (como por ejemplo replicación y encriptación).

Como hemos propuesto, trabajamos con un lenguaje que interpreta una aplicación desarrollada sobre la máquina abstracta, obteniendo así un lenguaje computacionalmente reflectivo. Basándonos en esta propiedad, podremos diseñar un *middleware* que facilite distintos niveles de persistencia para cualquier objeto, de forma transparente para el programador.

El software que implementa los distintos niveles de persistencia, modificará computacionalmente los objetos de una aplicación para que realicen llamadas al motor de la base de datos de forma automática, como se muestra en la figura 6. El programador deberá identificar únicamente el nivel de persistencia oportuno para los objetos de su aplicación.

Distintos niveles de persistencia realizarían actualización de objetos en, por ejemplo:

- La creación y destrucción de objetos.
- La invocación a un conjunto de métodos especificados.
- La modificación del estado de un objeto.
- Cada cierto intervalo de tiempo.

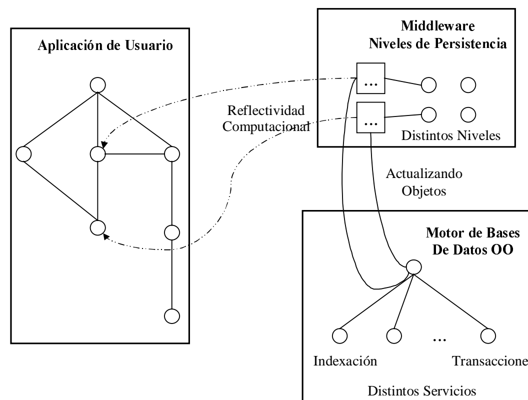

Fig. 6. Esquema de funcionamiento de un sistema de persistencia implícita.

Por ejemplo, si en tiempo de ejecución necesitamos hacer persistentes los objetos de una aplicación ya existente, podemos utilizar el sistema de persistencia implícita. Una vez ejecutándose la aplicación, se invoca en un nuevo proceso al *middleware* para que modifique el mecanismo de envío de mensajes de los objetos de la aplicación. Mediante reflectividad computacional éste añade al mecanismo del paso de mensajes las llamadas oportunas de actualización al motor de la base de datos para hacer persistir los objetos en caso de que hayan cambiado de estado. El código del usuario ni es necesario modificarlo, ni es consciente de que se está cambiando su funcionalidad.

Obviamente el nivel de persistencia elegido implicará una menor eficiencia cuando éste requiera un mayor número de actualizaciones. Por otro lado, el utilizar un software auxiliar descarga de trabajo y responsabilidad al programador, además de simplificar la depuración de sus aplicaciones.

5 Ventajas de la Arquitectura Presentada

La principal característica de este sistema de persistencia es la flexibilidad que proporciona su condición de reflectividad. Apoyándonos en esta característica, se pueden identificar varias ventajas frente a los sistemas ya existentes.

5.1 Uniformidad en la utilización del Sistema de Persistencia

El motor de la base de datos orientado a objetos es una parte del sistema integral. Será utilizado en diversas situaciones dentro del sistema: desde la línea de comandos del sistema operativo hasta una aplicación de usuario.

Los servicios del motor son utilizados de forma homogénea dentro del sistema, sin necesidad de dividir en archivos de datos, bases de datos, ficheros creados para configurar una aplicación, ficheros ejecutables,... Sólo existen objetos persistentes.

5.2 Funcionamiento Flexible

En las aplicaciones que utilicen la persistencia implícita, tenemos dos variables determinantes en tiempo de ejecución:

1. El nivel de persistencia identificado para los distintos objetos de la aplicación.
2. Las distintas técnicas o implementaciones posibles para cada servicio del motor.

La elección de estos dos parámetros influirá en el factor de actualización del sistema al almacenamiento secundario y en la eficiencia del mismo.

La flexibilidad otorgada por la reflectividad computacional permite modificar incluso los valores de las dos variables en función de ciertas necesidades a lo largo del tiempo. Por ejemplo, una aplicación puede utilizar distintos mecanismos de indexación en función de la carga del sistema; el sistema de persistencia puede ser desactivado si se requiere una ejecución rápida y activado posteriormente.

Esta flexibilidad necesita un control; no puede permitirse que de manera arbitraria se cambie el funcionamiento del sistema. Es necesario un mecanismo de protección que pueda emplearse de manera homogénea para controlar el paso de mensajes [23], utilizado particularmente y permitir de manera selectiva qué llamadas reflectivas pueden realizarse.

5.3 Sintonización de Parámetros

Encontrar el sistema óptimo de eficiencia en un sistema informático con recursos limitados es una tarea compleja. Esto es aplicable también a las bases de datos, que intentarán acceder a mayores volúmenes de información en el menor tiempo posible.

La aplicación de una serie de algoritmos de búsqueda, apoyándose en unas estructuras de datos no siempre tiene una evaluación absoluta. Unas estrategias pueden obtener mejores resultados que otras en un contexto y ser menos eficientes en otras situaciones. Así, por ejemplo Nested Index tiene el mejor rendimiento en operaciones de recuperación, sin embargo Multiindex tiene el mejor rendimiento en actualización [14]. Como hemos visto, necesitaremos estudiar la repercusión de una serie de variables influyentes en el motor de la base de datos y elegir una estrategia determinada en cada situación. Estos estudios pueden realizarse de forma muy sencilla en nuestro sistema gracias a la flexibilidad que lo caracteriza. Podemos identificar las siguientes variables:

- Clasificación de objetos dentro de una aplicación.
- Los distintos niveles de persistencia existentes.
- Las distintas implementaciones para cada servicio del motor.

Gracias a la reflectividad, es muy sencillo para una aplicación modificar las distintas variables emitiendo estadísticas para posteriormente llegar a un compromiso. Esto convierte este sistema en una plataforma ideal para realizar *benchmarking* entre diferentes técnicas de indexación, por ejemplo. Una vez seleccionados los criterios para la comparación, y en función de los resultados obtenidos en los diferentes estudios, se podrán establecer unas guías que permitan decantarse entre las distintas técnicas en función del tipo de aplicación más frecuente del sistema.

5.4 Alto Nivel de Abstracción en la Programación

Una vez implementado sobre el sistema varios niveles de persistencia como un *middleware*, la programación de aplicaciones será más sencilla puesto que elevamos el nivel de programación olvidándonos de la persistencia[4]. En este nuevo nivel de abstracción, debemos identificar los objetos que serán (o ya son) persistentes y el nivel de persistencia que deseamos darles.

Otra ventaja adicional de subir el nivel de abstracción utilizando un motor y el *middleware* que trabajará sobre él, es la facilidad a la hora de depurar, portar y mantener el software.

6 Trabajos Relacionados

Existen diversos lenguajes de programación reflectivos que se basan en el concepto de Protocolos de Meta-Objetos. El primero fue 3-KRS [16] y ejemplos actuales son OpenC++ [32] y MetaJava [28].

---

[4] Al sistema de persistencia hay que añadir la distribución, multitarea, seguridad y movilidad de objetos propios del sistema pero no abordados en este artículo.

Estos lenguajes posibilitan, o bien generando código adicional en tiempo de compilación, o bien ampliando el juego de instrucciones de una máquina virtual, una ejecución reflectiva *a priori*. No es posible modificar una aplicación que se está ejecutando, siendo necesaria la recompilación de los archivos fuentes. La característica de estos lenguajes es que sacrifican flexibilidad para conseguir una mayor eficiencia en tiempo de ejecución.

Basándose en estos meta-lenguajes se han creado prototipos de sistemas operativos orientados a objetos reflectivos como por ejemplo Apertos [33]. Cada objeto tiene su propio entorno de programación llamado meta-espacio. Para comunicarse con los meta-espacios, existen una serie de meta-objetos con funciones específicas.

Existen numerosos SGBDOO bien como servidores independientes (O2 y ObjectStore) o bien como complementos para lenguajes de programación (PSE y PSE Pro). Sin embargo, estos sistemas se comportan como cajas negras poco flexibles, no tienen la posibilidad de añadir dinámicamente nuevos mecanismos de indexación, que la aplicación pueda seleccionar el mecanismo de indexación que desea, etc. Por otro lado, estos sistemas están concebidos como elementos independientes y no como parte integral de un sistema global como se propone en este artículo.

En general en los SGBDOO actuales existen diferentes posibilidades de cara a conseguir la persistencia: *Por tipo* (un objeto puede ser persistente cuando se crea, basándose en su tipo, existiendo tipos persistentes y temporales como en Objectivity/DB), *por llamada explícita* (el usuario puede especificar de forma explícita la persistencia de un objeto, como en ObjectStore), y *por referencia* (la condición de persistencia se adopta a partir de determinados objetos persistentes existentes por alcance, como en GemStore). Sin embargo, un campo menos explorado es la utilización de la persistencia implícita a través de un mecanismo reflectivo en el sentido propuesto en este artículo, es decir, que permitan en tiempo de ejecución cambiar el grado de persistencia de un objeto, con el grado de libertad que eso conlleva para el programador.

Se ha creado un prototipo de persistencia dentro del proyecto Oviedo3 [22]. Su complejidad y limitaciones vienen dadas por partir del concepto erróneo de identificar la propiedad de persistencia como inherente a la máquina abstracta. La identificación de esta propiedad como código que utiliza la reflectividad es más sencilla y flexible.

7 Conclusiones

La programación de aplicaciones orientadas a objetos utilizando SGBDOO exige el conocimiento de distintas APIs y la inclusión de código adicional que opere con éstas. Una alternativa a este entorno de programación es la utilización de persistencia implícita en la que el usuario no necesita añadir funcionalidad a la aplicación codificando con código intruso las llamadas al sistema de persistencia. Se consigue así, solucionar problemas de complejidad, legibilidad y portabilidad del código.

El sistema de persistencia implícita es desarrollado sobre un sistema integral orientado a objetos, basado en una máquina abstracta que proporciona el soporte básico para los objetos del sistema. Un motor flexible de bases de datos es integrado de forma compacta en éste, proporcionando con ello una serie de beneficios. La flexibilidad del motor y del sistema de persistencia se consigue con la implementación de éste sobre la máquina abstracta dotada de reflectividad.

La programación tradicional de aplicaciones persistentes, se podrá realizar accediendo a las interfaces del motor de la base de datos desde el lenguaje de programación utilizando así persistencia explícita. Sin embargo, la implementación de un *middleware* que modifique computacionalmente cualquier aplicación permite establecer un sistema de persistencia implícita. Una aplicación en ejecución podrá ser modificada por este sistema para comunicarse con el motor estableciendo un nivel de persistencia de forma implícita.

Beneficios directos del sistema propuesto son la uniformidad en el uso del sistema de persistencia, un funcionamiento flexible para cualquier aplicación y la posibilidad de sintonizar y realizar estudios sobre el comportamiento de determinados parámetros en ciertos contextos de ejecución.

El resultado obtenido es un entorno de programación orientado a objetos de un elevado nivel de abstracción, portable y dotado de una alta flexibilidad, incluso en tiempo de ejecución.

Referencias

[1] R. Cattell . *Object Data Management. Object Oriented and Extended Relational Database Systems* (Revised Edition). Addison Wesley, 1994.

[2] R. Cattell, T. Atwood, J. Duhl et. al. *The Object Database Standard:ODMG-93*. Morgan Kaufmann, 1994.

[3] R. Cattell, D. Barry, D. Bartels. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 1997.

[4] M. Atkinson, L. Daynès, M. Jordan, T. Printezis, S. Spence. *An Orthogonally Persistent Java. SIGMOD Record, Vol 25 Nº4. December,1996*

[5] Steven T. Abell. *Using Java with PSE*. Java White Paper. http://www.odi.com, March 1999.

[6] *Jeevan User's Guide*. http://www.w3apps.com/, March 1999.

[7] *StreamStore User's Guide*. http://www.bluestream.com/ss/default.htm, March 1999.

[8] *Java Blend: Integrating Java Objects with Enterprise Data*. White paper. http://java.sun.com, March 1999.

[9] Darío Álvarez Gutiérrez. *An object-oriented abstract machine as the substrate for an object-oriented operating system.* 11th European Conference on Object-Oriented Programming (ECOOP'97). Jyväskylä (Finland). June 1997.

[10] A.B. Martínez, D. Álvarez, J.M. Cueva, F. Ortín , J.A. Pérez. *Incorporating an Object-*Oriented *DBMS into an Integral Object-Oriented System.* World Multiconference on Systemics, Cybernetics and Informatics and International Conference on Information Systems, Florida, 1998.

[11] E. Bertino and P. Foscoli. *Index Organizations for Object-Oriented Database Systems*. IEEE Transactions on Knowledge and Data Engineering. Vol.7, 1995.

[12] W. Kim, K.C. Kim, A. Dale. *Indexing Techniques for Object-Oriented Databases.* En W. Kim y F.H. Lochovsky (ed) : Object-Oriented Concepts, Databases, and Applications. Addison-Wesley, 1989.

[13] C. Chin, B. Chin, H. Lu. *H-trees: A Dinamic Associative Search Index for OODB.* ACM SIGMOD, 1992.

[14] E. Bertino and W. Kim. *Indexing Techniques for Queries on Nested Objects*. IEEE Transactions on Knowledge and Data Engineering. Vol.1 nº2, 1989.

[15] A. Kemper, C. Kilger and G. Moerkotte. *Function Materialization in Object Bases: Design, Realization, and Evaluation.* IEEE Transactions on Knowledge and Data Engineering, 1994.

[16] Pattie Maes. *Computational Reflection*. Technical Report 87_2, Artificial Intelligence Laboratory, Vrieje Universiteit Brussel, 1987.
[17] Jacques Ferber. *Computational Reflection in class based Object-Oriented Languages.* Proceedings of the Conference on Object-Oriented Programmings, Systems, Languages, and Applications, OOPSLA'89, New Orleans, Oct. 1989, pp. 317-326.
[18] Jacques Ferber. *Coceptual reflection and actor languages*. Meta-Level Architectures and Reflection. P. Maes, D. Nardi (Editors). North-Holland, 1988.
[19] *Java Core Reflection. API and Specification.* JavaSoft. January 1997.
[20] D. Álvarez, *Complete Persistence for an Object-Oriented Operating System using an Abstract Machine with Reflective Architecture*, Ph. D. Thesis, University of Oviedo, Spain, March 1998.
[21] J.M. Cueva. *The Integral Object Oriented System Oviedo3*. II Jornadas sobre Tecnologías Orientadas a Objetos. Oviedo, 1996.
[22] F. Ortín, D. Álvarez, R. Izquierdo, A.B. Martínez, J.M. Cueva. *The Oviedo3 Persistence System*. III Jornadas de Tecnologías de Objetos. Sevilla, 1997.(in spanish).
[23] M.A. Díaz, D. Álvarez, A. García-Mendoza , F. Álvarez, L. Tajes and J.M. Cueva**.** *Merging Capabilities with the Object Model of an Object-Oriented Abstract Machine.* Proceedings of the ECOOP'98 Workshop on Distributed Object Security and the 4th Workshop on Mobile Object Systems, pp. 9-13. Inria Rhône-Alpes, Francia, Julio de 1998.
[24] Glenn Krasner. *Smalltalk-80, bits of history, words of advise.* Xerox Palo Alto Research Center. Addison Wesley 1984.
[25] Pierre Cointe. *The OvjVlisp Kernel: a Reflective Lisp Architecture to define a Uniform Object-Oriented System.* Meta-Level Architectures and Reflection. P. Maes, D. Nardi (Editors). North-Holland, 1998.
[26] B.C. Smith, *Reflection and Semantics in a Procedural Language*, MIT-LCS-TR-272, MIT, Cambridge, 1982.
[27] Gregor Kizcales, J. des Rivieres, D.G. Bobrow.*The Art of Meta-Object Protocol.* MIT Press 91.
[28] Jügen Kleinöder, Michael Golm. *MetaJava: An Efficient Run-Time Meta Architecture for Java™.* TR-14-96-03. Computer Science Department, Friedrich-Alexander-University Erlangen-Nürnberg, Germany. June 1996.
[29] OMG. *CORBA: Architecture and Specification.* 1997.
[30] Chris Maeda, Arthur Lee, Gail Murphy, Gregor Kiczales. *Open Implementation and Design*. Proceedings Symposium on Software Reuse. May 1997.
[31] Douglas Kramer. *The Java™ Platform. A White Paper*. Sun JavaSoft. May 1996.
[32] Shigeru Chiba. *A Metaobject Protocol for C++*. Proceedings of the Conference on Object-Oriented Programmings, Systems, Languages, and Applications, OOPSLA'95. pp. 285-299.
[33] Y. Yokote. *Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach*. Workshop on Reflection and Meta-level Architectures at OOPSLA 93.