Conector ODBC para DES
ODBC Driver for DES

Trabajo de Fin de Grado
Curso 2024–2025

Autor
Sergio Miguel García Jiménez

Director
Fernando Sáenz Pérez

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid

Nota final: 10.0

Conector ODBC para DES
ODBC Driver for DES

Trabajo de Fin de Grado en Ingeniería Informática

Autor
Sergio Miguel García Jiménez

Director
Fernando Sáenz Pérez

Convocatoria: *Junio 2025*

Doble Grado en Ingeniería Informática y Matemáticas
Facultad de Informática
Universidad Complutense de Madrid
Nota final: 10.0

17 de junio de 2025

Agradecimientos

Agradezco a Fernando Saénz Pérez tanto por haber confiado en mí para desarrollar este componente de su propio ecosistema DES, como por su gran disposición como profesor en mi asignatura de Bases de Datos y como director de TFG.

Agradezco sobre todo a mamá, papá y Dani; por ellos soy yo y son a quienes me debo por completo.

Resumen

Conector ODBC para DES

El estándar ODBC (Open Database Connectivity), desarrollado por Microsoft y Simba, describe una API desde el estándar SQL para acceder a fuentes de datos, principalmente bases de datos relacionales. La mayoría de sistemas de gestión de bases de datos (e incluso otros sistemas como MS Excel) poseen su propio conector ODBC para acceso a sus fuentes de datos. Este trabajo presenta el desarrollo de un nuevo conector ODBC bautizado DESODBC para el sistema de bases de datos deductivas DES (Datalog Educational System) desarrollado por Fernando Sáenz Pérez, que soporta SQL entre otros lenguajes de consulta. Para el desarrollo se parte del conector ODBC open-source de MySQL (MyODBC), adaptándola adecuadamente a las características de DES. Finalmente, el conector ofrece versiones para Windows, GNU/Linux y macOS, que son las plataformas para las que DES está disponible.

Palabras clave

ODBC, DES, SQL, Datalog, MyODBC.

Abstract

ODBC Driver for DES

The ODBC (Open Database Connectivity) standard, developed by Microsoft and Simba, describes an API from the SQL standard to access data sources, mainly relational databases. The majority of the database management systems (and even other systems like MS Excel) provide their own ODBC Driver for accessing their own data sources. This work presents the development of a new ODBC driver named DESODBC for the database management system DES (Datalog Educational System) developed by Fernando Sáenz-Pérez, which supports SQL among other query languages. In the development we will start from the open-source ODBC Driver for MySQL (MyODBC), adapting it properly to the characteristics of DES. Finally, the driver offers versions for Windows, GNU/Linux y macOS, which are the platforms in which DES is supported.

Keywords

ODBC, DES, SQL, Datalog, MyODBC.

Índice

1. Introducción **1**

1.1. Antecedentes y motivación 1

1.1.1. ¿Qué es ODBC? 1

1.1.2. Datalog Educational System (DES) 2

1.1.3. Problema planteado 2

1.2. Objetivos 3

1.3. Plan de trabajo 4

1.4. Producto final 5

1.5. Organización de la memoria 6

2. Estado de la cuestión **7**

2.1. Antecedentes: DES y ODBC 7

2.2. Sobre el desarrollo de un conector ODBC, y cómo sería en DES . . . 8

2.2.1. Algunos aspectos preliminares específicos de DES 8

2.2.2. Por qué partimos del conector de MySQL 9

2.2.3. Esquema de DESODBC 10

2.2.4. Políticas de concurrencia 11

3. Descripción del trabajo **13**

3.1. Importación y compilación de MyODBC 13

3.2. Preliminares: fuentes de datos 14

3.3. Desarrollo del instalador 15

3.4. Desarrollo de la biblioteca de configuración 16

3.5. Desarrollo del conector ODBC 18

3.5.1. Clases principales y funciones tipo getter/setter 20

3.5.2. Funciones de entorno y conexión 23

3.5.3. Funciones de manejo de sentencias 43

3.5.4. Funciones de manejo de resultados 46

3.5.5. Funciones de catálogo (metadatos) 51

3.5.6. Sistema de gestión de errores 63

3.5.7. Versiones ANSI y Unicode del conector 65

3.6. *Testing* 67

3.7. Documentación 69
3.8. Sobre el desarrollo y depuración: software empleado 69
3.8.1. Independientes del sistema 69
3.8.2. Windows 70
3.8.3. Unix 73

4. Conclusiones y trabajo futuro **77**
4.1. Conclusiones 77
4.2. Trabajo futuro 77

Introduction **79**
4.3. Preliminaries and motivation 79
4.3.1. What is an ODBC? 79
4.3.2. Datalog Educational System (DES) 79
4.3.3. Proposed problem 80
4.4. Objectives 80
4.5. Work plan 82
4.6. Final product 83
4.7. Organization of the document 83

Conclusions and future work **85**
4.8. Conclusions 85
4.9. Future work 85

Bibliografía **87**

A. Instalación rápida del conector **95**

Capítulo 1

Introducción

1.1. Antecedentes y motivación

El presente trabajo constituye una implementación de un conector ODBC para el sistema de gestión de base de datos (en adelante, DBMS —*Data Base Management System*—) Datalog Educational System (DES).

1.1.1. ¿Qué es ODBC?

ODBC (Open Database Connectivity) es «*un estándar API para acceder a datos en sistemas de gestión de bases de datos tanto relacionales como no relacionales*» [1, pp. 3-4][1], de última versión 3.8 (2009) [2]. Este estándar fue desarrollado a principios de los años 90 por Microsoft y Simba Technologies, y resolvió un problema anteriormente impracticable: poder acceder bajo el estándar SQL a múltiples bases de datos desde una misma aplicación, sin programar código específico en ella para estas conexiones [1, p. 3].

De esta forma, un sistema de gestión de bases de datos (*Data Base Management System*, en adelante DBMS) podría disponer de su propio conector ODBC (o *ODBC driver*) que admita llamadas a funciones ODBC API para la comunicación con una de las fuentes de datos (o *data source*) del DBMS.

Así, una aplicación que quisiera conectarse a dicha fuente de datos del DBMS solo necesitaría comunicarse con un ODBC Driver Manager (por ejemplo, en Windows, `System32/odbc32.dll`) que actúe de intermediario entre la aplicación y el conector. De esta forma, la comunicación entre la aplicación y la fuente de datos deseada tendría lugar sin necesidad de programación extra; esto es, agnóstica a la implementación del conector y a la naturaleza de la fuente de datos o su DBMS asociado.

[1] Traducción mía sobre el original en inglés.

1.1.2. Datalog Educational System (DES)

Datalog Educational System (DES) es un sistema de bases de datos deductivo open-source, multiplataforma y gratuito publicado originalmente en 2004 por Fernando Sáenz Pérez, profesor titular de la Universidad Complutense de Madrid [3]. DES, entre muchas de sus características, soporta consultas (*queries*) en lenguajes *Datalog*, *SQL*, *álgebra relacional* (RA, *Relational Algebra*), *cálculo relacional basado en tuplas* (TRC, *Tuple Relational Calculus*) y *cálculo relacional basado en dominios* (DRC, *Domain Relational calculus*) [4].

El desarrollador, que además se encarga del mantenimiento hasta el día de hoy, tuvo como objetivo principal el uso de DES en el ámbito educativo para estudiantes de bases de datos, con el fin de enseñar de forma intuitiva e interactiva los conceptos detrás de una base de datos deductiva, así como enseñar los lenguajes soportados en consultas que hemos enumerado [4, p. 10]. Sin embargo, su uso se extendió en universidades de todo el mundo, llegando a tener como efecto colateral incluso su uso en investigación para diferentes campos [5].

El autor del presente Trabajo de Fin de Grado fue uno de estos estudiantes a los que se les introdujo el sistema, teniendo como profesor al propio desarrollador en la asignatura de Bases de Datos del tercer curso del doble grado de Ingeniería Informática y Matemáticas; este profesor es, además, el tutor de este TFG.

1.1.3. Problema planteado

El director del presente trabajo de fin de grado describió el problema planteado en las siguientes líneas textuales:

> *«ODBC (Open Database Connectivity) es una API estándar para el acceso a fuentes de datos, principalmente bases de datos relacionales, que fue desarrollada por Microsoft y Simba. Actualmente existen conectores (drivers) ODBC a la mayoría de bases de datos (MySQL, PostgreSQL, DB2...) e incluso a otras fuentes como Excel y ficheros CSV. En este proyecto se desarrollará un nuevo conector ODBC para el sistema de bases de datos deductivas DES (`http://des.sourceforge.net`), que incorpora varios lenguajes de consulta sobre estructuras de datos relacionales. El lenguaje de desarrollo habitual para estos conectores es C, y se puede partir de la documentación de Microsoft e implementaciones open-source ya desarrolladas tales como PostgreSQL y MySQL. Dado que DES no soporta consultas concurrentes, será necesario implementar en el propio conector su gestión. Dependiendo del número de alumnos se podrán desarrollar conectores no solo para Windows, sino también para Linux y macOS.»* [6]

En efecto, si bien DES es capaz de conectarse a un conector ODBC para comunicarse con otros DBMS, DES carece de su propio conector ODBC. Esto es, no existe un conector mediante el cual un ODBC Driver Manager pueda comunicarse con DES, de forma que aplicaciones externas puedan conectarse a bases de datos específicas de este DBMS.

1.2. Objetivos

Una vez presentado el problema planteado, el objetivo de este trabajo de fin de grado es desarrollar este conector ODBC, entendiendo que sus características fundamentales serán: (1) admitir llamadas efectuadas por el ODBC Driver Manager, que demanda cierta información; (2) acceder a las bases de datos DES por medio de su propia API ya implementada (*Textual API* o TAPI, como veremos), y (3) devolver la información requerida al ODBC Driver Manager. Para ello, partiremos del código fuente de MyODBC, el conector ODBC de MySQL.

Vamos a implementar un conector ODBC de la última versión de ODBC (3.8) para la última versión de DES (V6.7). Nos planteamos como objetivo mínimo concreto ofrecer un conector ODBC para DES en todas las versiones de sistema operativo que DES soporta: Windows, GNU/Linux y macOS.

Al partir del conector MyODBC ya implementado para el DBMS MySQL, también nos vamos a plantear como objetivos opcionales implementar todas las funcionalidades que ofrece su *suite*, a saber:

- Ejecutable instalador capaz de:
  - Registrar el conector ODBC en el sistema.
  - Añadir y actualizar fuentes de datos.
- Configurador GUI (*Graphical User Interface*) en Windows para los parámetros de una fuente de datos asociada al conector ODBC[2].
- El conector ODBC en sí, con soporte para:
  - Sistemas operativos Windows, GNU/Linux con unixODBC, macOS con unixODBC e iODBC[3].

[2]MyODBC provee en su código fuente medios para compilar bibliotecas de configuración GUI en GNU/Linux. Estas bibliotecas no se distribuyen en las versiones oficiales de MyODBC, por lo que no incluimos como requisito el proporcionar configuradores GUI para GNU/Linux. Sin embargo, se trabajó sobre estas bibliotecas extra-oficiales y las hemos incluido en el trabajo final con algunas consideraciones (véase la sección 3.4).

[3]En realidad, el conector MyODBC ofrece soporte para otros sistemas basados en Unix en general incluyendo Oracle Solaris. DES puede usarse en este último sistema operativo si se compila tanto DES como las dependencias SWI-Prolog/SICStus desde sus respectivos fuentes. DES no presenta una versión oficial para Oracle Solaris y, por tanto, nos restringimos entonces a los sistemas operativos sobre los que DES ofrece soporte oficial: Windows, GNU/Linux, macOS.

No obstante, creemos que, salvo alguna posible cuestión menor a solventar, este conector puede compilarse y funcionar en Oracle Solaris. Esto se debe a que, como se verá, nuestra implementación es prácticamente agnóstica a los sistemas basados en Unix a los que esté destinado, con excepción de algunas variaciones relativas a funciones POSIX.

- Formatos ANSI y Unicode[4].

En resumen, cumpliremos con el objetivo mínimo pero aspiraremos asimismo a lograr estos últimos objetivos opcionales. Con esto, cumplimos sobradamente con los objetivos propuestos por el director del TFG en la presentación del problema (sección 1.1.3).

Con la implementación de este *software* habremos revisitado todos los conceptos *software* que se ven en el actual plan de estudios: tratamos programación orientada a objetos, bases de datos, estructuras de datos, algoritmos, políticas de concurrencia, mecanismos IPC (*Inter-Process Communication*), diseño de interfaces gráficas (y sus respectivos criterios de buenos diseños de sistemas interactivos), corrección, cuestiones internas de sistemas operativos, uso de herramientas para desarrollo de *software*, etc.

1.3. Plan de trabajo

1. Fase de investigación (*Julio 2024 - Agosto 2024, Enero 2025 - Marzo 2025*). Los interrogantes a responder, son:

   - Qué es ODBC.
   - Qué es un conector ODBC.
   - Cómo es el flujo de uso de un conector ODBC y agentes involucrados.
   - Qué funcionalidades de DES podemos utilizar.
   - Qué funcionalidades tiene MyODBC.
   - Qué entiende exactamente MyODBC por ANSI y Unicode.
   - Cómo funciona a bajo nivel una conexión ODBC en Windows y sistemas basados en Unix.
   - Qué funcionalidades de Windows API y POSIX podemos utilizar.

   Parte de estos interrogantes serán respondidos e investigados mientras dure nuestro periodo de desarrollo y nos vayamos encontrando con retos no previstos. Es por esto que la fase de investigación y la de desarrollo son tareas amplias y generales que se entrelazan y se solapan en el tiempo.

2. Fase de desarrollo (*Julio 2024 - Agosto 2024, Enero 2025 - Marzo 2025*). El flujo que hemos considerado más cómodo es el siguiente:

   a) Importación del código fuente de MyODBC y compilación en Windows.
   b) Proceso paralelo de adaptación del conector a las necesidades de DES, y eliminación de dependencias de biblioteca externa relativas a MySQL.
   c) Adaptación para los sistemas basados en Unix GNU/Linux y macOS.

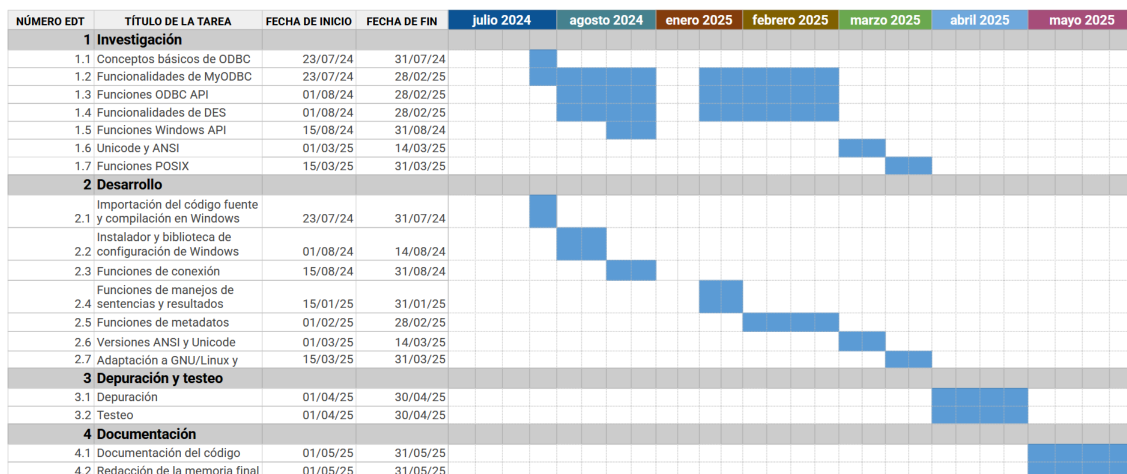
[4]En el apartado 3.5.7 estudiaremos a qué nos referiremos exactamente.

3. Fase de testeo y depuración (*Abril 2025 - Mayo 2025*).

   - Pruebas con casos de uso tanto típicos como «patológicos» para depurar posibles errores de concepto, bugs, y fallas de seguridad.
   - Depuración y refinamiento del conector, más allá del que se realizó durante el desarrollo del mismo.

4. Redacción de la memoria final y documentación del código (*Abril 2025 - Mayo 2025*).

El diagrama de Gantt que muestra el flujo de tareas concretas y desgranadas puede verse en la figura 1.1.



NÚMERO EDT	TÍTULO DE LA TAREA	FECHA DE INICIO	FECHA DE FIN
1	**Investigación**		
1.1	Conceptos básicos de ODBC	23/07/24	31/07/24
1.2	Funcionalidades de MyODBC	23/07/24	28/02/25
1.3	Funciones ODBC API	01/08/24	28/02/25
1.4	Funcionalidades de DES	01/08/24	28/02/25
1.5	Funciones Windows API	15/08/24	31/08/24
1.6	Unicode y ANSI	01/03/25	14/03/25
1.7	Funciones POSIX	15/03/25	31/03/25
2	**Desarrollo**		
2.1	Importación del código fuente y compilación en Windows	23/07/24	31/07/24
2.2	Instalador y biblioteca de configuración de Windows	01/08/24	14/08/24
2.3	Funciones de conexión	15/08/24	31/08/24
2.4	Funciones de manejos de sentencias y resultados	15/01/25	31/01/25
2.5	Funciones de metadatos	01/02/25	28/02/25
2.6	Versiones ANSI y Unicode	01/03/25	14/03/25
2.7	Adaptación a GNU/Linux y	15/03/25	31/03/25
3	**Depuración y testeo**		
3.1	Depuración	01/04/25	30/04/25
3.2	Testeo	01/04/25	30/04/25
4	**Documentación**		
4.1	Documentación del código	01/05/25	31/05/25
4.2	Redacción de la memoria final	01/05/25	31/05/25

Figura 1.1: Diagrama de Gantt asociado a nuestra planificación temporal.

1.4. Producto final

El autor del presente trabajo terminaría alcanzando todos los objetivos propuestos, incluyendo los opcionales: además de la implementación de Windows, la de GNU/Linux y macOS. En los objetivos propuestos por el director se explica que este escenario de exportar a distintos sistemas operativos hubiera aplicado en el caso de múltiples alumnos. Explicamos en este punto que, como veremos, el código de las funcionalidades en sí del conector es directamente exportable para todos los compiladores y sistemas operativos; lo único que habrá que tener en cuenta para cada sistema es la cuestión de la conexión a un proceso DES y los aspectos IPC (*Inter-Process Communication*) implicados, así como aspectos sobre herramientas auxiliares (bibliotecas de configuración, como veremos). Es por esto que el autor de este trabajo consideró que el trabajo extra a realizar no era el suficiente como para requerir un equipo.

Nuestro producto final es un conector ODBC, de nombre DESODBC, y cuyo repositorio de GitHub donde se aloja se encuentra en el siguiente enlace: `https://github.com/segarc21/des-connector-odbc`.

1.5. Organización de la memoria

La presente memoria describe en el capítulo 3 tanto el desarrollo de la fase de investigación como el de implementación en forma paralela con fines didácticos y expositivos. Las conclusiones se describen en el capítulo 4.

Capítulo 2

Estado de la cuestión

2.1. Antecedentes: DES y ODBC

Un esquema ilustrativo en el que se despliega la arquitectura de ODBC puede verse en la figura 2.1.

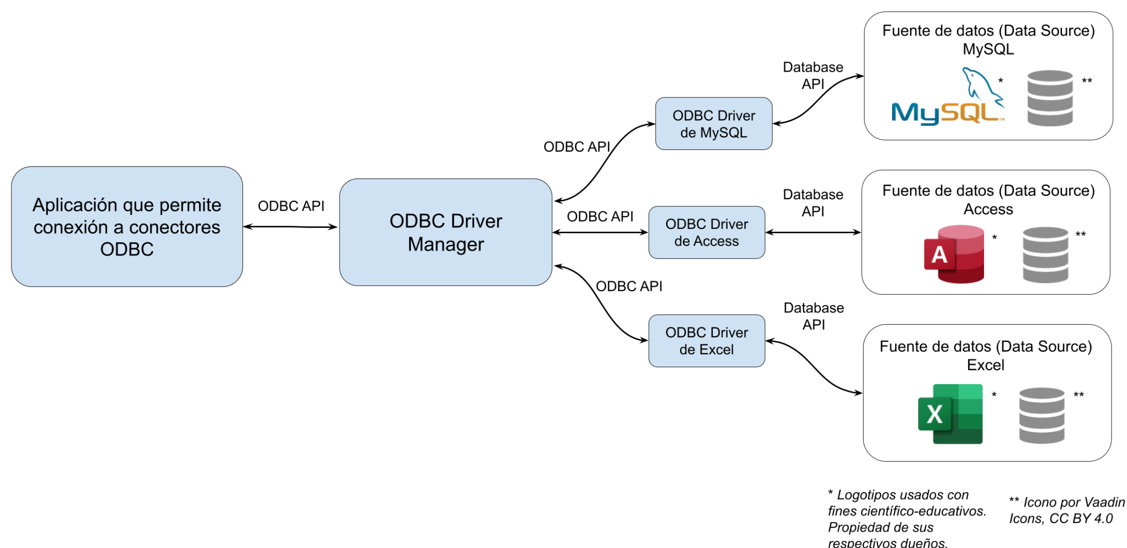


Figura 2.1: Esquema ejemplo del modelo ODBC.

Desarrollando algo más la exposición de la motivación, DES puede realizar conexiones a bases de datos externas por medio del estándar ODBC: esto es, DES puede comportarse a nivel de aplicación según el diagrama de la figura 2.1, y puede conectarse a cualquier fuente de datos, como bases de datos cuyo DBMS soporte conexiones ODBC (MySQL, MS Access, Oracle, ...) [7, p. 13]. Las conexiones se realizan por medio del comando `/open_db` (véase la figura 2.2), con la posibilidad de cerrarlas por medio del comando `/close_db`[1].

Como adelantamos previamente, DES carece sin embargo de su propio conector ODBC, y su implementación constituye el objetivo del presente trabajo.

[1]Más información sobre el soporte ODBC se encuentra en [7, sección 5.1].

```
DES> /open_db mysql
DES> select * from edge
answer(a:integer(4),b:integer(4)) ->
{
    answer(1,2),
    answer(2,3),
    answer(3,4)
}
Info: 3 tuples computed.
```

Figura 2.2: Ejemplo de conexión al conector ODBC de MySQL [7, sección 5.1.8].

2.2. Sobre el desarrollo de un conector ODBC, y cómo sería en DES

2.2.1. Algunos aspectos preliminares específicos de DES

Primero, señalar que no debemos preocuparnos de desarrollar la capa *Database API* planteada en la figura 2.1, pues DES implementa una API mediante la cual una aplicación externa puede comunicarse con sus bases de datos, presentando la información que se requiera en un formato de fácil análisis. Esta API fue bautizada por el autor de DES como TAPI *(Textual API)*, y se activa añadiendo el prefijo `/tapi` a las consultas que enviemos a DES [7, sección 5.18].

Extendámonos un poco en el funcionamiento de TAPI, pues sobre esta API se basará la entrada y salida entre nuestro conector y DES.

Nuestro conector podrá leer y escribir sobre un proceso DES usando las corrientes (*streams*) estándares de entrada, salida y error. Sin embargo, analizar desde un programa una salida que está escrita en formato legible para los humanos resulta complicado. Enviar la consulta con el prefijo `/tapi` permite obtener una salida sobre la cual es mucho más sencillo extraer la información desde algoritmos escritos en nuestro conector. En la figura 2.3 ilustramos la diferencia entre obtener una salida en formato humano y en formato TAPI.

Tal y como puede verse en dicha figura, el formato TAPI presenta una salida más fácil de analizar para un programa informático. Tras una operación `SELECT`, la salida empieza con la palabra clave `answer`. El carácter `$` constituye el delimitador entre las distintas listas de atributos (en el ejemplo, en orden: nombres y tipos de columnas, primera columna, segunda columna) y `$eot` indica el final de la salida. Existen distintos formatos de salida TAPI según el comando, y en el manual de DES [7, sección 5.18] puede encontrarse la especificación completa de TAPI.

```
DES> select * from people

answer(people.id:int,
       people.name:string) ->
{
  answer(23,'anderson'),
  answer(57,'nolan')
}
Info: 2 tuples computed.
```

```
DES> /tapi select * from people
answer
people.id
int
people.name
string
$
23
'anderson'
$
57
'nolan'
$eot
```

Figura 2.3: Comparación entre consultas estándar y las TAPI.

Por último, hemos de notar que un conector ODBC para DES no podrá explotar todas las funcionalidades de este sistema. Esto se debe a que DES no es un mero DBMS relacional, sino que es un sistema de gestión de base de datos deductivo ampliamente polivalente, que en particular ofrece aspectos de bases de datos relacionales como posibilidad (DES implementa el estándar SQL). Por tanto, nuestro conector no podrá lidiar de forma «nativa» con todos los aspectos relacionados con `Datalog`, puesto que se sale del ámbito ODBC. Nuestro conector ofrecerá las funciones API de ODBC, por lo que solo se ofrecerán funciones relativas a lo que engloba el estándar SQL.

No obstante, con nuestro conector, de alguna manera se puede trabajar con todos los aspectos de DES al existir la opción de enviar a la consola de DES comandos directos (con funciones API ODBC de tipo *execute*) y visualizar la correspondiente salida, lo que posibilita explotar todas las funcionalidades de DES desde la aplicación que se conecte al conector ODBC.

2.2.2. Por qué partimos del conector de MySQL

Desarrollar un conector ODBC para un DBMS desde cero es una tarea de gran complejidad y que requiere conocimientos avanzados de distintos aspectos a considerar en el conector (tipos de datos, codificación de caracteres, concurrencia, compatibilidades, toda la casuística de errores a mostrar al usuario, etcétera). De esta forma, tal y como planteó el director del trabajo, consideramos que lo inteligente es estudiar un conector ODBC ya existente e irlo adaptando progresivamente a las particularidades de DES.

Para ello, hemos optado por el conector MyODBC (*MySQL Connector/ODBC*), pues es posible realizar su estudio, modificarlo y distribuirlo al ser un conector de código abierto (*GPLv2*) y ser MySQL uno de los DBMS más populares del mundo, con las facilidades que aquello implica para solucionar posibles incidencias en nuestro desarrollo.

Primero, vamos a ver las características de DES y cómo debe comportarse nuestro conector de acuerdo con estas.

2.2.3. Esquema de DESODBC

Cuando una aplicación como MySQL, Access, o cualquier otra quiere conectarse a un ODBC, primero carga en su espacio de memoria una copia del ODBC Driver Manager (ODBC Driver Manager en Windows; unixODBC en sistemas basados en Unix como GNU/Linux, macOS y Oracle Solaris; y también iODBC para macOS)[2].

Cuando la aplicación desea conectarse a un conector ODBC, aloja espacio para un objeto conexión (que veremos después) y pide al ODBC Driver Manager que se conecte a la fuente de datos especificada. El ODBC Driver Manager procede a cargar en el espacio de memoria de la aplicación una copia de la biblioteca que corresponde al conector.

Ahora surge la cuestión de cómo plantear la arquitectura. Se nos abren muchas posibilidades según el grado de compartición de recursos y visibilidad que deban tener las distintas conexiones: desde que exista una instancia global de DES para todas las conexiones, hasta el otro extremo en el que cada conexión abra un proceso DES.

Sin embargo, nuestra arquitectura vendrá determinada por el siguiente problema: existe un conflicto cuando dos aplicaciones distintas abren procesos DES desde un mismo directorio de trabajo. En estas circunstancias existe un conflicto con el fichero de estado, que es un fichero que alude al estado de la base de datos local de DES. Por ejemplo, cuando se llame en DES al comando `/save_state`, se guarda el estado en un fichero `des.sds` en el directorio de trabajo desde el que se ha inicializado DES; y en general, por defecto, los comandos que recibe DES son ejecutados desde rutas relativas al directorio de trabajo. Es por esto que aplicaciones distintas que pretendan trabajar en un mismo directorio de trabajo necesariamente se deban conectar a un único proceso DES compartido. Optamos por abrir un proceso DES por cada conexión que lo lance desde un nuevo directorio de trabajo[3] [4].

[2]En el caso de Windows, cuando las aplicaciones llaman al .dll correspondiente a nuestro conector, carga una copia del mismo (Windows carga instancias de las bibliotecas en el espacio de direcciones de la aplicación que alude a estas bibliotecas [8]). Ocurrirá lo mismo con todas las bibliotecas DLL, en particular también con nuestro conector.

Los análogos a DLL para GNU/Linux y MacOS (.so, .dylib) presentan en este punto el mismo comportamiento de carga con copia.

[3]Algo similar se hace en DESweb, aplicación del ecosistema DES que ofrece acceso a DES en `https://desweb.fdi.ucm.es/`. El servidor subyacente ejecuta un único proceso DES por cada directorio de trabajo, que alude al directorio generado para un cliente que se conecta.

[4]Aunque haya sido explicado de forma implícita, aclaramos que esto aplica incluso si las aplicaciones se refieren a ejecutables DES distintos. En general, DES se instalaría en un único directorio, pero también podría instalarse en varias rutas distintas por alguna necesidad del usuario: en ese caso también habría conflicto si aludimos a ejecutables DES distintos en un mismo directorio de trabajo, pues los distintos procesos escribirían en el mismo directorio de trabajo.

Así, cuando una aplicación B pretenda conectarse a un directorio de trabajo ya usado por otra conexión A, si el proceso global DES preexistente asociado a ese directorio de trabajo alude a

Ilustramos el esquema resultante en la figura 2.4 en el que se representa el caso de Windows (análogo en lo fundamental para sistemas basados en Unix).

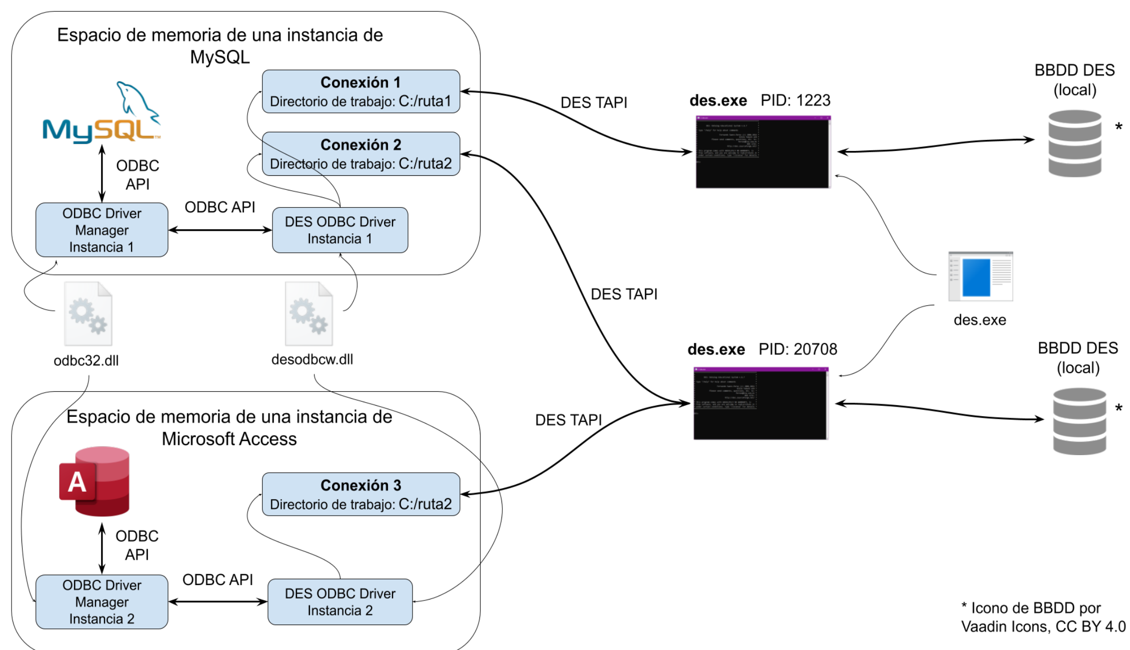


Figura 2.4: Esquema ilustrativo del funcionamiento de DES ODBC.

De esta manera, como vemos en esta figura 2.4, MySQL realiza dos conexiones distintas a DES desde dos directorios de trabajo distintos; ergo se abren dos procesos distintos. No obstante, Access se conecta a DES desde un directorio de trabajo que resulta que comparte otra conexión realizada por MySQL (`C:/ruta2`) y por tanto, procede a compartir dicho proceso ya abierto con MySQL.

2.2.4. Políticas de concurrencia

El hecho de que DES sea un DBMS de los denominados embebidos o locales (esto es, se integran directamente en la aplicación y no precisan de tratamiento en red, como por ejemplo *SQLite*) facilita el desarrollo de la concurrencia. Además, DES no provee transacciones, niveles de aislamiento y otros mecanismos de seguridad al no ser un DBMS deductivo completo [7, p. 10].

Sin embargo, el hecho de que sea posible que dos conexiones distintas aludan a un mismo proceso DES nos obliga a implementar algunas políticas de concurrencia. En lo que al propio DES se refiere, no tendremos que considerar ningún aspecto de concurrencia, puesto que DES no tiene soporte para transacciones, y desde los procesos DES se practican lectura/escritura de archivos del sistema (por medio de comandos `COMMIT`, `ROLLBACK` y otros especiales de DES) sin ninguna concurrencia más allá de la que implementa el propio sistema operativo.

un ejecutable distinto al que B requiere, no será posible la conexión de B (distintos ejecutables inevitablemente involucrarían distintos procesos). Se le notificará a B que debe aludir a la ruta del ejecutable usando en la conexión de A.

Sin embargo, con lo que respecta a conexiones que hagan uso de un mismo proceso DES, será fundamental implementar mecanismos de concurrencia seguros y efectivos para resolver lo que sería a priori un problema de *readers/writers* (*R/W*) con respecto a la entrada/salida de la consola asociada a un proceso compartido DES.

La implementación de la concurrencia entre las instancias del conector y este único proceso DES se basará en memoria compartida. Las instancias del conector accederán a los datos que permitan comunicarse con el único proceso DES (una única serie de tuberías) planificándolos con políticas sencillas de concurrencia, de grano muy grueso. Esto se debe a que nuestro problema R/W viene simplificado: cada conexión simplemente adquiere un cerrojo, que durará en su posesión durante todo el proceso completo de consulta (envío de consulta y recepción del resultado), después de lo cual procede a liberar el cerrojo.

Esto tiene la ventaja, además, de que las siguientes conexiones que pretendan hacer nuevas consultas no puedan leer comandos ejecutados previamente por otros, puesto que la tubería de lectura se consume dentro de cada proceso de intercambio de información. Es por esto que nuestro problema R/W no es el problema R/W clásico considerado en literatura informática.

Nuestra implementación de la concurrencia involucra más detalles de los que hemos tratado en esta sección. Veremos más adelante todas las consideraciones que tuvimos en cuenta.

Capítulo 3

Descripción del trabajo

En este capítulo se describe todo el trabajo realizado. El orden en el que se hace no pretende seguir el del plan de trabajo en el apartado 1.3: en este capítulo, con fines expositivos y didácticos, trataremos de ir describiendo de forma paralela e hilvanada conceptos teóricos de ODBC y aspectos de implementación. Tras haber descrito el desarrollo del conector, incluimos secciones para describir el *software* empleado, *testing* y documentación.

3.1. Importación y compilación de MyODBC

La versión de MyODBC empleada fue la 9.0.0, y como primer paso la importamos y compilamos de manera directa en Windows.

No procede describir con detalle los pasos de importación y compilación. Sin embargo, la tarea en sí fue relativamente compleja por los numerosos retos que atravesamos para compilar un *software* de esta complejidad. MyODBC proporciona documentación para compilar el fuente en Windows (véase [9]), pero sin embargo tuvimos nosotros que lidiar con la cuestión de la compilación en modo `Debug` del proyecto. El problema surge cuando las dependencias externas (como la de MySQL Server, `mysqlclient.lib`), están compiladas en modo `Release`. Por tanto, hemos tenido que compilar las dependencias en modo `Debug`, tarea engorrosa pues estas dependencias a su vez en ocasiones dependían de otras.

Es por esto que empleamos un trabajo y tiempo considerables para compilar el proyecto de MyODBC en Windows en modo `Debug` antes de empezar a trabajar con el mismo. Sin embargo, una vez desarrollado nuestro conector y habiendo eliminado todas las dependencias de biblioteca externa, el proyecto es autocontenido. De esta manera, el proyecto se exportó a GNU/Linux y macOS compilándolos con simples comandos de `cmake`[1], sin necesitar dependencia externa alguna.

[1]Como nota, no existen distintos proyectos para los distintos sistemas operativos. Esto se debe al uso de directivas de compilador `#ifdef _WIN32`, `#ifdef _UNIX_`, `#ifdef __APPLE__`, etcétera; cada compilador en el sistema operativo conocerá entonces qué código debe compilar y qué código de otros sistemas operativos debe ignorar. Esto permite tener un único código fuente sobre el que compilar universal a todos los sistemas operativos objetivo.

3.2. Preliminares: fuentes de datos

Un conector ODBC constituye un único archivo de biblioteca (`.dll` en Windows, `.so` en sistemas basados en Unix y también `.dylib` en macOS).

Veamos el proceso de registro de un conector en nuestro sistema aprovechando para introducir conceptos a los que aludiremos continuamente. Los siguientes conceptos son compartidos por todos los sistemas operativos en los que ODBC tiene soporte.

El concepto fundamental es el de fuente de datos (*data source*) u origen de datos. Una fuente de datos consiste, de manera abstracta, en un registro de sistema configurado por el usuario del conector (en adelante en todo el trabajo, «el usuario») a través del cual se acceden a unos determinados datos. Estos registros de sistema (bajo un nombre) tienen como atributos unos determinados parámetros que permiten la conexión a dichos orígenes de los datos. De esta manera, podemos registrar fuentes de datos asociadas a sistemas de gestión de bases de datos, o también para otros sistemas como Excel (que ofrece orígenes de datos como archivos `.CSV`).

Por ejemplo, una fuente de datos asociada a MySQL tiene como atributos la ruta del conector ODBC asociado sobre el que realizar la conexión, nombre de usuario y contraseña, y otros parámetros relativos a configuración relativa a la red, conjuntos de caracteres, etcétera. De esta manera, la fuente de datos es la vía de entrada a las bases de datos que se encuentren asociadas a esa cuenta-usuario en MySQL, y que establece que las conexiones se realicen bajo las condiciones que se establecen en dicha fuente de datos.

Para el caso de DES, el concepto de fuente de datos correspondería a una base de datos local DES (recuérdese el esquema en la figura 2.4). De esta forma, ya podemos advertir que nuestras fuentes de datos asociadas a DES deberán referenciar de alguna manera, y de forma mínima e indispensable, las rutas del conector DESODBC, del ejecutable DES, y del directorio de trabajo.

Siguiendo con la explicación del concepto de fuente de datos, también por ejemplo podríamos querer tener dos fuentes de datos asociadas al conector de DES, solo que uno trabaje desde un cierto directorio de trabajo A, y el otro desde otro directorio de trabajo B[2]. Ambos, tanto conceptualmente como en el sistema, corresponden a fuentes de datos distintas. Por ejemplo, una aplicación como MySQL podría optar por conectarse a DES mediante solo la fuente de datos que corresponde a la del directorio de trabajo A.

Antes de crear las fuentes de datos tenemos que tener registrado nuestro conector para que la fuente de datos pueda aludir a él. Esto se hace en el archivo `odbcinst.ini`[3], donde se registra un conector ODBC especificando, como mínimo, la ruta de su archivo de biblioteca. Dependiendo del conector, se incluyen más pará-

[2]No obstante, nada nos impide registrar dos fuentes de datos idénticas, de mismos atributos, solo que nombrados de manera distinta.

[3]En Windows constituye un archivo de registro. En sistemas basados en Unix, son ficheros normales alojados en determinados sitios, según la instalación del usuario.

metros asociados al registro del conector. En nuestro caso, los atributos del conector que hemos considerado son:

- `DRIVER`: la ruta de la biblioteca del conector (obligatorio).
- `SETUP`: la ruta de la biblioteca de configuración (veremos más adelante qué es esto).

Una vez registrado el conector, creamos una fuente de datos asociada a él registrándolo en el fichero `odbc.ini`[4]. Las fuentes de datos pueden ser de tipo usuario, en cuyo caso `odbc.ini` tiene permisos de usuario, o de sistema, que puede requerir permisos elevados. Lo más común es que existan distintos archivos `odbc.ini` en rutas distintas dependiendo de si aluden a fuentes de datos de usuario o por el contrario del sistema.

En las primeras líneas del archivo `odbc.ini` se declaran las fuentes de datos: nombre, y a qué conector registrado alude. Dentro del registro de la fuente de datos en sí, los atributos que se requieren y que hemos considerado necesarios son:

- `Driver`: la ruta de la biblioteca del conector. Realmente no es un atributo imprescindible puesto que ya podemos conocerlo por medio de saber a qué conector está asociado, pero es estándar incluirlo.
- `DESCRIPTION`: descripción que se quiera poner a esta fuente de datos.
- `DES_EXEC`: ruta del ejecutable DES.
- `DES_WORKING_DIR`: ruta del directorio de trabajo requerido desde el que se ejecutará `DES_EXEC`.

3.3. Desarrollo del instalador

El instalador es un binario ejecutable de tipo línea de comandos de gran sencillez. Se ofrecen comandos para instalar el conector, instalar una fuente de datos con atributos específicos, listar las fuentes de datos disponibles, etcétera. Hemos podido reutilizar el código de esta aplicación casi por completo, eliminando la posibilidad de incluir los atributos relacionados con MySQL e implementando los que necesita una fuente de datos de nuestro conector. En la figura 3.1 podemos ver un ejemplo de su uso.

[4] Mismas consideraciones que con `odbcinst.ini`.

```
  shell> desodbc-installer -d -a -n "DES ODBC Unicode Driver"
  -t "DRIVER=C:\desodbcw.dll;SETUP=C:\desodbcS.dll"
Success: Usage count is 1

shell> desodbc-installer -s -a -n "DES ODBC Test Data Source"
  -t "DRIVER=DES ODBC Unicode Driver;
      DES_EXEC=C:\des\des.exe;DES_WORKING_DIR=C:\des"
Success
```

Figura 3.1: Ejemplo de instalación del conector y de una fuente de datos.

Dedicaremos el apéndice A para explicar cómo realizar una instalación rápida de nuestro conector y de una fuente de datos en el sistema.

3.4. Desarrollo de la biblioteca de configuración

Habíamos mencionado de pasada que al registrar nuestro conector en el sistema, podíamos especificar una biblioteca de configuración. MyODBC, y por consiguiente nuestro conector, incluye esta biblioteca cuya función es modificar los atributos de un *data source* registrado, con la atractiva posibilidad añadida de poder llamar a esta biblioteca en tiempo de ejecución de una aplicación cliente (`SQLDriverConnect`, como veremos), para poder especificar y/o cambiar los atributos de la fuente de datos dinámicamente antes de conectarse al mismo.

Recalcamos que siempre tenemos la posibilidad de cambiar los atributos de la fuente de datos por medio de modificar `odbc.ini`, pero la biblioteca de configuración nos ofrece el poder hacerlo por medio de una interfaz gráfica (GUI): véase como ejemplo la GUI que hemos realizado para Windows en la figura 3.2.

Procedimos a adaptar el proyecto de la biblioteca de configuración en Windows. La biblioteca está diseñada en base a un archivo de recursos (*resource file*) (véase [10]) que especifica la GUI en sí, y una serie de archivos fuente C++ que interaccionan con ella. Se modifican estos archivos convenientemente, ayudándonos del editor visual de archivos `.rc` proporcionado por Visual Studio, y el resultado final puede verse en la figura 3.3.

Con respecto a la biblioteca de configuración para los sistemas basados en Unix, es importante señalar que no se ofrecen este tipo de bibliotecas para estos sistemas. Desarrollarlas excede los objetivos propuestos en la sección 1.2, donde declaramos que MyODBC ofrecía una biblioteca de configuración solo para Windows.

No obstante, recordemos que siempre tenemos la posibilidad de modificar los archivos `odbcinst.ini` y `odbc.ini` manualmente en cualquier sistema y que de hecho, suele ser lo estándar y esperable para el usuario. En macOS con iODBC también existe la posibilidad de acceder al *iODBC Data Source Administrator*, que permite editar de manera mínimamente gráfica los atributos de cualquier fuente de datos.

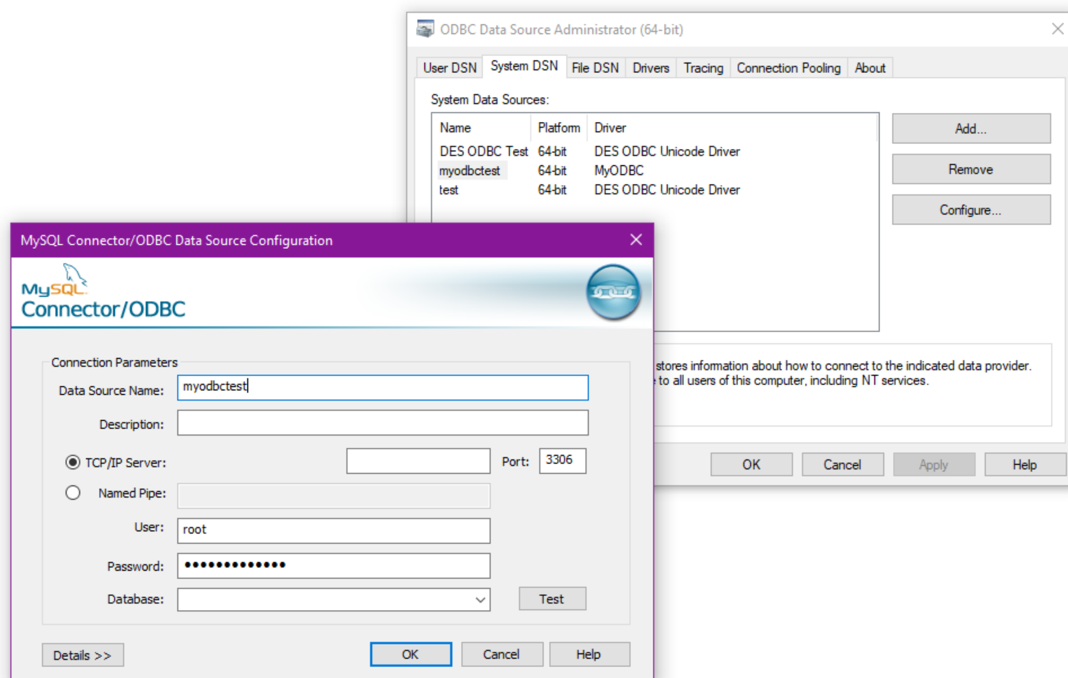


Figura 3.2: Interfaz gráfica que provee la biblioteca de configuración de MyODBC. Al pulsar en configurar una fuente de datos en el *ODBC Data Source Administrator* (Windows), se llama a la biblioteca de configuración mostrando al usuario una GUI sobre la cual puede modificar los atributos de la fuente de datos.

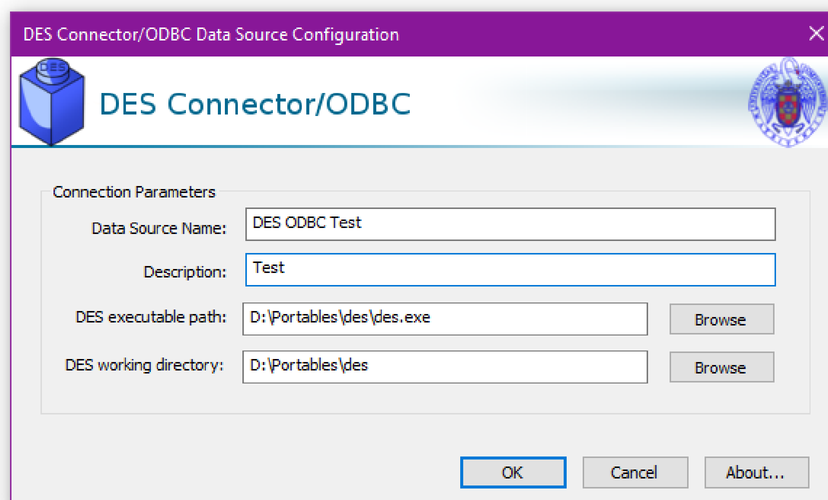


Figura 3.3: Interfaz gráfica que provee la biblioteca de configuración de DESODBC en Windows.

Sin embargo, pese a que MyODBC no ofrezca en sus distribuciones oficiales bibliotecas de configuración para GNU/Linux, el código fuente permite compilar tres de ellas. Hemos trabajado sobre estas bibliotecas de configuración intentando rescatarlas: sin embargo, no ofrecemos garantías sobre su funcionamiento correcto al igual que MyODBC tampoco. Tras una serie de pruebas, los productos obtenidos y sus competencias son:

- Una biblioteca «minimalista» que hemos bautizado `libdesodbcS.so` (en el conector de MySQL, `libmyodbcS.so`), que puede verse en la figura 3.4. Esta biblioteca ofrece la lectura y escritura de parámetros con una función estándar `ODBCINSTGetProperties` a la cual una determinada aplicación debe llamar. unixODBC refiere en su web a una *suite* de aplicaciones externas `unixodbc-gui-qt`[5] que proporciona a unixODBC una interfaz GUI basada en Qt [11], pero que debe descargarse y compilarse a mano. Constatamos que `libdesodbcS.so` trabaja correctamente sobre esta *suite*.

- Dos bibliotecas GUI, una de versión GTK-2 y la otra GTK-3, llamándolas `libdesodbcS-gtk2.so` y `libdesodbcS-gtk3.so` bajo la misma política de nombres que antes. Véase la figura 3.5. Estas bibliotecas no funcionan correctamente bajo `unixodbc-gui-qt`, y en trabajos futuros se considerará el proveer esta compatibilidad. No obstante, una aplicación puede ejecutar estas bibliotecas cuando llama a `SQLDriverConnect` requiriendo completar información —el comportamiento esperado consiste en mostrar la interfaz de la biblioteca de configuración antes de realizar la conexión—.

De esta forma, pese a que no garanticemos oficialmente la corrección de estas bibliotecas, pueden resultar de utilidad en los contextos que hemos mencionado.

Con respecto a la descripción del trabajo, para modificar la GUI en Windows tuvimos que modificar un archivo de recursos (*resource file*); el procedimiento análogo aquí será modificar un archivo `.glade` que especifica en formato XML el aspecto de la GUI. Dicho formato corresponde al tipo de archivo que genera la herramienta Glade, editor visual que permite desarrollar interfaces gráficas de tipo GTK. Por tanto, con Glade editamos este archivo y rediseñamos la GUI buscando el aspecto lo más similar posible a nuestro resultado en Windows, y finalmente exportamos a GTK-2 y GTK-3.

3.5. Desarrollo del conector ODBC

Como ya adelantamos en la introducción de este capítulo, describiremos la implementación hilvanando con los conceptos teóricos necesarios para comprenderla. El manual que define el estándar y sobre el que conoceremos los distintos elementos de ODBC es la documentación de Microsoft [2].

La implementación de un conector ODBC, en esencia y al tratarse de una biblioteca, consiste en la implementación de las funciones de la API ODBC. Nuestra

[5]https://github.com/mazbrili/unixodbc-gui-qt

filosofía en este documento será pasar por las funciones más importantes (no procede describir aquí las de poco interés, que son numerosas), y que clasificaremos según su ámbito de uso.

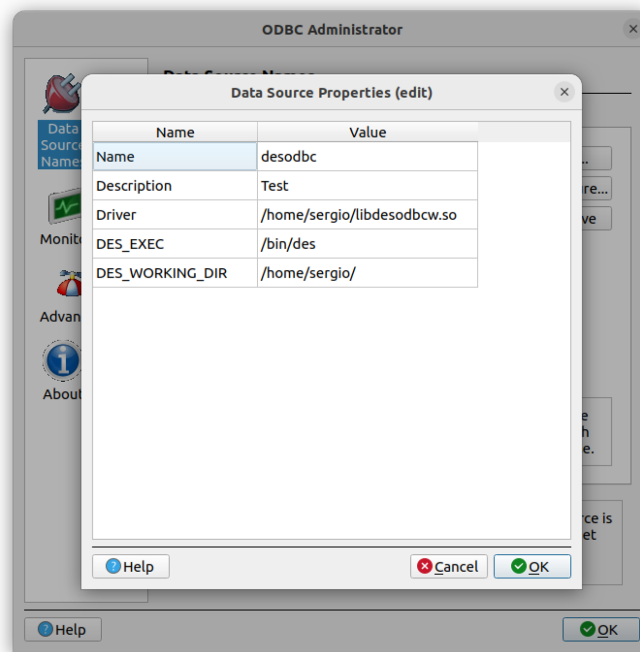


Figura 3.4: `libdesodbcS.so` (desde `ODBCManageDataSourcesQ5`). Esta aplicación administradora de fuentes de datos llama a la función `ODBCINSTGetProperties` de `libdesodbcS.so`, y recoge los resultados obtenidos en una GUI (ventana `Data Source Properties (edit)`) proporcionada por la propia aplicación.

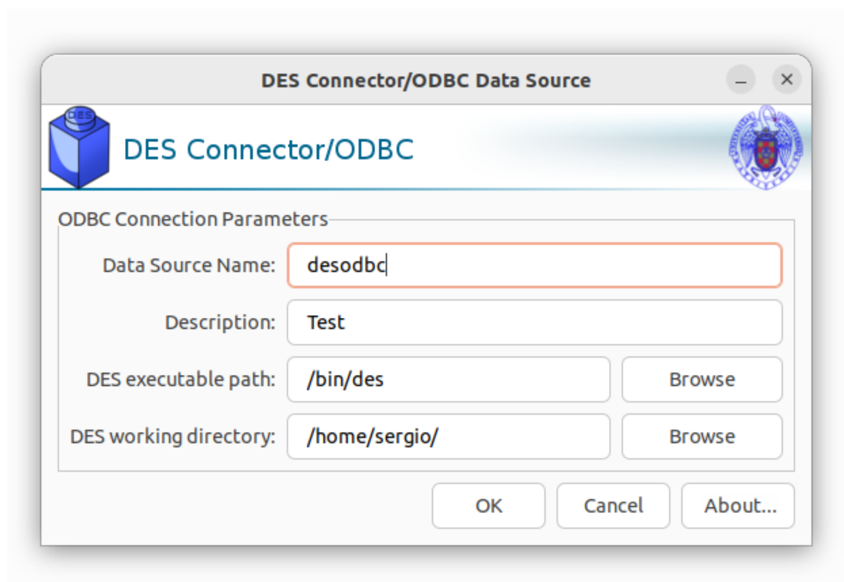


Figura 3.5: `libdesodbcS-gtk2.so` / `libdesodbcS-gtk3.so` ejecutada tras una llamada a `SQLDriverConnect`.

En el estándar ODBC no es obligatorio implementar todas las funciones: ODBC ofrece un abanico amplio de funciones a las que se puede referenciar, pero cada DBMS posee unas características únicas que puede hacer que ciertas funciones no tengan sentido en algún contexto. Esto ocurre en MyODBC y también en el caso de DESODBC: hay funciones que, como se verá en cada caso, no hemos implementado por las circunstancias pertinentes. El estándar será que cuando el usuario referencia a estas funciones, se obtenga un mensaje de «no implementado».

Asimismo, describiremos solo los detalles más relevantes de las implementaciones.

3.5.1. Clases principales y funciones tipo getter/setter

3.5.1.1. Elementos básicos de ODBC. Manejadores

Procedemos a exponer brevemente los conceptos teóricos fundamentales en ODBC, a los que nos referiremos continuamente durante la memoria, y que son fundamentales para entender el comportamiento esperado de las funciones del estándar.

La API ODBC define una serie de elementos (o en adelante pensando en términos de implementación «clases») que todo conector ODBC debe considerar en su implementación: estructura de entorno (*environment*), conexión (*connection*), sentencia (*statement*) y descriptores (*descriptors*) [12, p. 87], que ahora pasamos a explicar.

Cada conector implementa los atributos que considere oportuno en las clases para ajustarse a las características del DBMS objetivo; la aplicación espera simplemente de cada clase el comportamiento adecuado para los estándares ODBC.

Por tanto, como el ODBC Driver Manager y las aplicaciones son agnósticas a la implementación interna del conector —los elementos son cajas negras fuera del conector— todas las partes implicadas —aplicación, ODBC Driver Manager, conector— consensúan la manipulación de unos elementos llamados manejadores (*handles*), que son punteros de 32 bits que apuntan a regiones del mapa de memoria del conector ODBC donde cada respectiva instancia de la clase está albergada [13].

Listamos los manejadores del estándar en orden de más global a más específico, siguiendo la estructura y partiendo de las explicaciones de [12, pp. 87-91]:

Manejadores de entorno (*Environment handles*) El elemento de entorno constituye una clase que albergan atributos de carácter global, tal y como información relativa al sistema operativo o versión de ODBC que se emplea. En el estándar suelen ser elementos con pocos atributos y muy generales.

En nuestro caso el elemento entorno se materializará en una clase C/C++ denominada `ENV`.

La clase de entorno guarda una lista de sus manejadores de conexión (*connection handles*) asociados, correspondientes al manejador que se describe a continuación.

Manejadores de conexión (*Connection handles*) La clase conexión alberga atributos relacionados con una conexión determinada. Esto es lo que correspondía

a los objetos «conexión» que ya vimos en la figura 2.4[6]. En DBMS que hacen uso de redes estos podrían ser el puerto, el tamaño máximo de los paquetes... Otros ejemplos serían el conjunto de caracteres empleado, registros (*logs*), base de datos actual a la que se apunta, información sobre el último error, etcétera.

Para futuras referencias, en nuestro caso el elemento conexión se materializará en una clase C/C++ denominada `DBC`.

La clase conexión guarda una lista de sus manejadores de sentencia (*statement handles*) asociados, correspondientes al manejador que se describe a continuación.

Manejadores de sentencia (*Statement handles*) Hacen referencia a la clase que guarda atributos relativa a una sentencia (*statement*), y así considera toda la información relativa al proceso completo de una consulta; la cadena de la consulta en sí, parámetros utilizados, resultados arrojados por el DBMS, etcétera. A los resultados que acabamos de mencionar se les llama conjuntos resultado (*result sets*).

Aclaramos en este punto que la cadena que el usuario envía como consulta son simplemente instrucciones válidas de acuerdo con el lenguaje del DBMS objetivo; en nuestro caso, comandos soportados por DES.

Para futuras referencias, en nuestro caso el elemento sentencia se materializa en una clase de C/C++ denominada `STMT`.

Una clase sentencia guarda una lista de sus manejadores de descriptor (*descriptor handles*) asociados, correspondientes al manejador que se describe a continuación.

Manejadores de descriptor (*Descriptor handles*) Un descriptor es un objeto de una clase que alberga metadatos respecto a los objetos sentencia [12, p. 91].

En este punto es necesario introducir brevemente los conceptos de *buffer* y parámetros.

La aplicación que se conecta al conector puede declarar una serie de variables que puede pedir al conector enlazar a una sentencia, de forma que se actualicen automáticamente al ejecutar una consulta o llevar a cabo ciertas operaciones con él. El estándar que se utiliza como variable son estos *buffers*, concretamente los buffers de datos (*data buffer*) y buffers de longitud (*length/indicator buffer*) [14] (véase cómo se organizan en la figura 3.6). Se suele hacer referencia a ellos como buffers de aplicación (*application buffers*) porque son creados en la aplicación.

Por ejemplo, si una aplicación quiere enlazar una variable `employee_name` para que el conector, al ejecutar una consulta, escriba sobre la variable con la información de la columna `name` del resultado final conforme se navegue por las filas, la aplicación debe proporcionarle estos *buffers* de datos y longitud para que, al concluir la operación, sepa recuperar la cadena final a partir de las direcciones del buffer sobreescrito y la longitud final (direcciones que han sido respetadas por el conector).

[6] Tal y como se infiere de la figura referenciada, los elementos *entorno* no juegan ningún papel relevante. Lo único que hacen es albergar los objetos conexión, que son los que realmente participan en la arquitectura de nuestro conector. Lo ortodoxo es que una aplicación solo aloje un objeto entorno, pero puede alojar un número arbitrario de ellos. En nada cambiará que un objeto conexión se encuentre alojado en un objeto entorno o en otro. Lo único importante es el directorio de trabajo al que alude cada conexión.

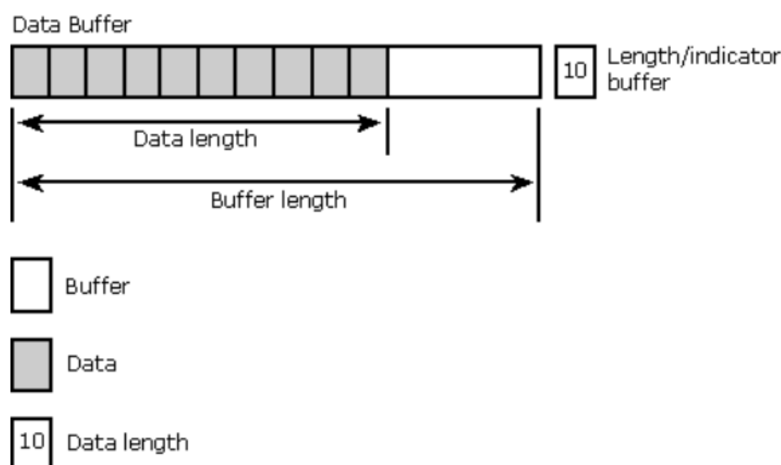


Figura 3.6: Arquitectura de los *buffer* en el estándar ODBC. Extraído de [14].

Los parámetros, por otra parte, son las variables que acabamos de definir pero enlazadas a parámetros-variable en una sentencia ODBC. Por ejemplo, podemos querer ejecutar una consulta [15]

```
INSERT INTO employee(name, age) VALUES (?, ?)
```

enlazando los dos parámetros con dos respectivas variables de la aplicación que ya tengan escrita la información que nos interesa.

Estamos en posición de definir los cuatro tipos de descriptores definidos en la API. Presentamos citas textuales de [12, p. 91] por su concisión[7].

- *Application Parameter Descriptor (APD)*: «*contiene información relativa a los buffers enlazados por la aplicación a los parámetros de una sentencia*».

- *Implementation Parameter Descriptor (IPD)*: «*contiene información sobre los parámetros asociados a la sentencia*», ya no a nivel aplicación sino desde la semántica específica del DBMS.

- *Application Row Descriptor (ARD)*: «*contiene información sobre los buffers de aplicación enlazados a las columnas del conjunto de resultados*». Nótese que enlazamos variables a columnas: para terminar de determinar el valor de la variable, se usa la fila actual (atributo del objeto sentencia) que la aplicación puede manipular a su antojo por medio de llamadas al conector.

- *Implementation Row Descriptor (IRD)*: «*contiene información sobre las columnas presentes en el conjunto de resultados de la sentencia*».

Para futuras referencias, en nuestro caso los elementos descriptores serán objetos de una estructura en C/C++ denominada `DESC`.

[7] Traducciones propias al castellano.

3.5.1.2. Funciones tipo *getter/setter*

Si bien hemos declarado que los elementos constituyen cajas negras para los agentes que no son el conector, sí que se espera que alberguen una serie de atributos mínimos que la aplicación puede obtener y modificar. Las funciones getter/setter son el procedimiento clásico que respeta la encapsulación y el concepto de caja negra; además, consecuentemente, los atributos en cuestión son devueltos en el formato de *buffers* que define el estándar. La función API ODBC `SQLGetInfo` es una función que permite conocer qué valores concretos nuestro conector o la fuente de datos en la que estemos posee acerca de ciertos atributos predefinidos en el estándar (por ejemplo, máximo número de usuarios conectados al mismo tiempo) [16]. También encontramos las siguientes funciones en los siguientes ámbitos:

Diagnóstico Hacen referencia a los errores: `SQLGetDiagRec`, `SQLGetDiagField`, `SQLError`.

Entorno Funciones `SQLSetConnectAttr`, `SQLGetConnectAttr`.

Sentencia Funciones `SQLSetStmtAttr`, `SQLGetStmtAttr`.

Conexión Funciones `SQLSetConnectAttr`, `SQLGetConnectAttr`.

Descriptores Funciones `SQLCopyDesc`, `SQLGetDescField`, `SQLSetDescField`, `SQLGetDescRec`, `SQLSetDescRec`.

Ha sido preciso modificar alguna de estas funciones que venían implementadas en MyODBC, por ejemplo en relación a escenarios en los que alguno de estos atributos hacen referencia a capacidades que MySQL y DES no comparten (por ejemplo, una aplicación podría llamar a `SQLGetConnectAttr` pidiendo el `SQL_LOGIN_TIMEOUT`, concepto que no tiene sentido en un DBMS local/embebido como DES).

3.5.2. Funciones de entorno y conexión

3.5.2.1. Entorno

`SQLAllocHandle` es una función que ordena al conector que aloje espacio en memoria para cierto tipo de elemento (conexión, entorno, sentencia o descriptores) y devuelva su puntero (el manejador) [12, p. 140]. Esta función simplemente llama a los constructores de las clases correspondientes y por tanto ha podido ser reutilizada en su totalidad.

3.5.2.2. Conexión

El proceso de conexión es el primer punto en el que MyODBC diferiría de forma fundamental a DESODBC, puesto que MySQL es un DBMS en red y DES es un DBMS local/embebido.

La función paradigmática para conectarnos a una fuente de datos asociada a un conector es `SQLConnect`. En el listado 3.1 podemos ver su cabecera.

Observamos que la aplicación cliente (MySQL, Access...) previamente primero ha alojado espacio para un manejador de entorno, y posteriormente para un manejador de conexión (`ConnectionHandle`) asociado a este manejador de entorno.

```
1 SQLRETURN SQLConnect(
2      SQLHDBC ConnectionHandle,
3      SQLCHAR * ServerName,
4      SQLSMALLINT NameLength1,
5      SQLCHAR * UserName,
6      SQLSMALLINT NameLength2,
7      SQLCHAR * Authentication,
8      SQLSMALLINT NameLength3);
```

Listado 3.1: Cabecera de la función `SQLConnect`. Extraído de [17].

Vemos que recibimos en `SQLConnect` parámetros de servidor (`ServerName` y `NameLength1`), usuario (`UserName` y `NameLength2`) y autenticación (normalmente la contraseña) (`Authentication` y `NameLength3`). El nombre del servidor corresponde al nombre de la fuente de datos instalada en el sistema. DESODBC ignora los parámetros de usuario y autenticación; no existe ninguna forma de autenticación en DES.

La implementación de `SQLConnect` en DESODBC es inicialmente idéntica a la de MyODBC, y consiste en la recuperación de los atributos guardados en el sistema relativos a la fuente de datos especificada (en caso de que la fuente de datos no exista, el propio ODBC Driver Manager se encarga de lanzar el error correspondiente).

Una vez todos los atributos se han almacenado en una pertinente clase `DataSource` que hemos reutilizado parcialmente de MyODBC, se procede al proceso de conexión.

Otra función de conexión del API ODBC es `SQLDriverConnect`. Esta función es en esencia igual a `SQLConnect`, solo que asiste previamente al usuario cuando este precisa información o ha incluido la insuficiente para realizar la conexión [18]. Como ya adelantamos antes, en nuestro caso en estos escenarios se lanzará la ventana GUI asociada a nuestra biblioteca de configuración, con lo que podremos rellenar de manera dinámica los atributos `DES_EXEC` y `DES_WORKING_DIR` en el tiempo de ejecución de la aplicación cliente (de esta forma, podremos especificar estos atributos sin escribir en el registro del sistema operativo).

Por último, la otra función de conexión del API es `SQLBrowseConnect`, que asiste al usuario de una forma más exhaustiva, pero su implementación es compleja. MyODBC opta por no hacerlo; en nuestro caso, tampoco tendría sentido hacerlo teniendo en cuenta además la sencillez de nuestras fuentes de datos.

Con respecto a la conexión en sí, no podemos reutilizar nada de la conexión al servidor de MySQL del sistema, puesto que el proceso de conexión a DES es completamente distinto. La tarea consiste entonces en reescribir completamente una función

interna `connect` que es la que se encarga de la conexión en sí. El funcionamiento a grandes rasgos común a todas las versiones de DESODBC (tanto Windows como los sistemas basados en Unix GNU/Linux y macOS) se muestra en el algoritmo 3.1.

Caso I. La aplicación llamadora es el único usuario conectado actualmente al directorio de trabajo (i.e., el número de usuarios conectados es cero previamente a la llamada).

1. Creamos las tuberías (*pipes*) que nos proporcionarán la comunicación bidireccional con un proceso global DES que ahora nos dispondremos a crear.
2. Procedemos a crear el proceso global de DES que proveerá servicio a todas las aplicaciones que se conecten a DESODBC con mismo directorio de trabajo:
   a) Ahora que ya conocemos todos los atributos asociados a la fuente de datos, consultamos el atributo `DES_EXEC` que constituye la ruta al ejecutable DES en el sistema.
   b) Creamos un proceso DES a partir de la ruta, y se le asocian las tuberías que hemos creado sobre su entrada y salida.
   c) Se consume el mensaje inicial que DES proporciona en su inicio.

Caso II. La aplicación llamadora no es el primer usuario conectado actualmente al directorio de trabajo (ya existe un proceso global DES ejecutándose desde ese directorio de trabajo).

1. Nos conectamos a las tuberías que ya fueron previamente creadas por el primer usuario.

Algoritmo 3.1: Proceso de conexión a DES en alto nivel común a todos los sistemas.

Los usuarios pueden conocer si son el primer usuario o no por medio de leer cierto atributo de la memoria compartida. La memoria compartida será la forma en la que las distintas conexiones del conector se comuniquen con el fin de conocer el estado global del proceso DES, el número de usuarios actualmente conectados, etc.

El acceso a la memoria compartida parte de la siguiente idea: nuestra aplicación identifica la ruta del directorio de trabajo examinando la fuente de datos, y la aplica un *hashing* de forma que obtenemos un identificador único para ese directorio de trabajo. Los nombres de los archivos en memoria compartida se construyen con este identificador como sufijo, de forma que cada conexión sabe a qué archivos específicos debe acceder y solo a esos. Véase la figura 3.7.

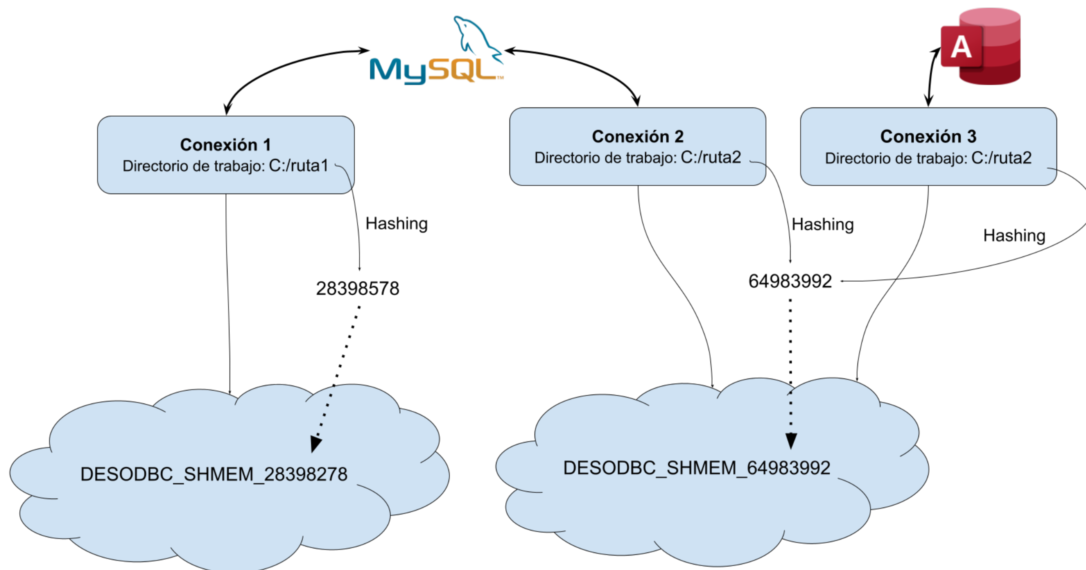


Figura 3.7: Esquema del *hashing* para acceso a memoria compartida.

Los preliminares al algoritmo 3.1 consisten en conocer el número de usuarios conectados. Recordemos que para acceder a memoria compartida, solo podemos conocer los nombres de archivo cuando aplicamos el *hashing* al directorio de trabajo. Por ello, en el proceso de conexión, primero recogemos la información de la fuente de datos. Acto seguido, construimos los nombres de los ficheros IPC de memoria compartida y cerrojos (véase el listado 3.2).

```
1  // driver/connect.cc (DESODBC)
2  this->connection_hash =
3        std::to_string(str_hasher(des_exec_path_str + des_working_dir_str));
4
5     this->SHARED_MEMORY_NAME = buildName(SHARED_MEMORY_NAME_BASE);
6     this->SHARED_MEMORY_MUTEX_NAME = buildName(
          SHARED_MEMORY_MUTEX_NAME_BASE);
7     this->QUERY_MUTEX_NAME = buildName(QUERY_MUTEX_NAME_BASE);
8     this->IN_WPIPE_NAME = buildName(IN_WPIPE_NAME_BASE);
9     this->OUT_RPIPE_NAME = buildName(OUT_RPIPE_NAME_BASE);
```

Listado 3.2: Construcción de nombres de los archivos IPC.

En este punto es preciso hablar del mecanismo IPC para la concurrencia. Como puede inferirse del código anterior, nuestro sistema de exclusión mutua es bastante sencillo, con únicamente dos cerrojos: el cerrojo de memoria compartida, y el cerrojo de consulta.

Anteriormente, cuando tratamos la cuestión de las políticas de concurrencia, ya adelantamos que debíamos implementar una solución de exclusión mutua de grano grueso. Para realizar una consulta, las aplicaciones competirán por un único cerrojo; para leer y escribir en memoria compartida, lo mismo para otro único cerrojo.

Se nos presenta, tanto en la memoria compartida como en la consola de DES, dos problemas *readers/writers* (R/W). Una de las soluciones más refinadas y eficientes al problema de los R/W es la que se conoce como solución por paso de testigo (véase [19, sección 4.4.3]), cuya implementación es de una complejidad conceptual considerable. Eso permite lecturas paralelas en ciertas condiciones, mejorando la eficiencia, cosa que en DESODBC no consideraremos puesto que supone problemas de consistencia. Más en detalle, nuestra política de grano grueso obedece a los siguientes motivos:

1. Nuestro modelo de la entrada y salida de DES no se parece a una base de datos sobre la que leer o escribir. Un usuario debe únicamente leer las salidas correspondientes a los comandos que él mismo ha enviado, y ninguno más. Es necesario entonces adquirir un único cerrojo para el proceso completo de envío de consulta y lectura del resultado, con el objeto de consumir la tubería de salida antes de liberar el cerrojo.

2. Nuestra memoria compartida tampoco atiende exactamente al modelo de base de datos. No se trata de un simple *array* sobre el que operar como supone el modelo general R/W[8]; la memoria compartida incluye una serie de atributos los cuales tienen relación y coherencia lógica entre sí de ciertas maneras muy concretas, y por tanto la única manera de evitar estados inconsistentes es que las instancias adquieran un único cerrojo de memoria compartida, privando a otros de leer y escribir, con libertad y el tiempo que ellas necesiten hasta que finalicen las tareas correspondientes y la memoria compartida vuelva a ser consistente. Más adelante veremos alguno de estos procesos.

En cuanto a la implementación concreta de los cerrojos, en la versión de GNU/-Linux no abriremos ficheros semáforo correspondientes a los cerrojos (sí en Windows y macOS, aunque el primero en RAM y el otro en disco), sino que estarán dentro de la memoria compartida. Esto es lo más deseable para no dejar archivos en el sistema que luego nos tenemos que preocupar de borrar. Sin embargo, esto no será posible en macOS: aunque macOS sea basado en Unix, en particular no implementa la función POSIX `sem_init` que posibilitaría guardar unos semáforos sin nombre en memoria compartida: no nos queda más remedio que usar semáforos con nombre que se almacenan en disco, usando `sem_open` y derivados.

Volviendo a nuestro proceso de inicialización del IPC, tras construir los nombres procedemos a inicializar estos objetos, que se almacenarán en la estructura conexión (`DBC`). Usamos los procedimientos correspondientes de la tabla 3.1, que lista las funciones IPC empleadas en los diferentes sistemas.

[8]En realidad, para el caso de Windows la memoria compartida sí que poseerá un *array* de clientes conectados, pero esto es solo uno de todos los atributos que la estructura de memoria compartida presenta. Como el número de clientes conectados tiende a ser muy bajo y acotado por 256, no merece la pena implementar una solución R/W eficiente y por ello complicar bastante innecesariamente el código para este único atributo *array* tan pequeño de entre todos los atributos. En efecto, los clientes son pocos y dicho *array* se lee y escribe en momentos muy concretos (solo en procesos de conexión y desconexión de clientes, que ocurren separados en el tiempo en el orden

Procedimiento	**Unix**	**Windows**
Crear/abrir región de memoria compartida	`shmget`	`CreateFileMapping` (`winbase.h`)
Adaptar memoria compartida a estructura predefinida	`shmat`	`MapViewOfFile` (`memoryapi.h`)
Crear tubería	`mkfifo`	`CreatePipe` (`namedpipeapi.h`)
Abrir tubería	`open`	—
Crear cerrojo	`sem_open`/`sem_init`	`CreateMutexEx` (`synchapi.h`)
Adueñarse del cerrojo	`sem_wait`	`WaitForSingleObject` (`synchapi.h`)
Liberar cerrojo	`sem_post`	`ReleaseMutex` (`synchapi.h`)
Cerrar cerrojo	`sem_close`	`CloseHandle` (`handleapi.h`)
Cerrar tubería	`close`	`CloseHandle` (`handleapi.h`)
Cerrar descriptor de fichero/manejador	`close`	`CloseHandle` (`handleapi.h`)
Cerrar vista a memoria compartida	`shmdt`	`UnmapViewOfFile` (`memoryapi.h`) y `CloseHandle` (`handleapi.h`)
Destruir región de memoria compartida	`shmctl`	`UnmapViewOfFile` (`memoryapi.h`) y `CloseHandle` (`handleapi.h`)
Destruir cerrojo	`unlink`	`CloseHandle` (`handleapi.h`)

Tabla 3.1: Procedimientos IPC empleados en los distintos sistemas.

El uso de las funciones Unix difieren en algunas ocasiones en GNU/Linux y macOS, como acabamos de adelantar. Si bien antes habíamos introducido el concepto de manejadores y descriptores en el contexto de ODBC, en el contexto de los sistemas operativos adquirirán significados diferentes. El uso del término *manejador* aludirá en este contexto a los *handle* de Windows (puntero a un recurso de Windows), y el término *descriptor* a los descriptores de fichero de Unix. Como puede observarse en la tabla anterior, los manejadores de Windows y los descriptores de fichero en Unix serán conceptos homólogos.

Como nota, en Unix hemos usado `shmget` y derivados en lugar de `shm_open`, pues así situamos nuestra estructura de memoria compartida en RAM en lugar de

de magnitud de segundos, minutos u horas) y cada operación sobre el array es prácticamente instantánea al ser tan pequeño.

en disco, y accedemos con una clave que en nuestro caso corresponderá al *hashing* (mayor anonimidad).

Por lo demás, la implementación concreta de los procesos descritos en el algoritmo 3.1 varían en gran medida dependiendo de si estamos en la versión de Windows o Unix. Procedemos a describir la implementación específica de ambas situaciones, comentando qué retos se nos presentaron y por qué se tomaron algunas decisiones concretas en base a las opciones que existían.

Caso Unix (GNU/Linux, macOS) La implementación tanto para GNU/Linux como para macOS son muy similares. Esto se debe a que llamamos a funciones POSIX que son soportadas por estos sistemas (en macOS, no todas, como ya hemos aclarado), y en consecuencia las funciones IPC que usaremos —relativas a memoria compartida, semáforos, tuberías con nombre (*named pipes*)— varían poco entre ellos.

Detengámonos un momento en por qué hemos elegido tuberías con nombre. El hecho de que hayamos escogido tuberías con nombre en lugar de tuberías sin nombre (*anonymous pipes*) responde, primero, a la sencillez: es muy fácil la comunicación bidireccional entre procesos con tuberías con nombre en sistemas de tipo Unix en un problema de nuestras características. La solución final será una arquitectura muy próxima a lo que se estudia en este plan de estudios de informática, por ejemplo en la asignatura de Sistemas Operativos.

Lo cierto es que pese a ello, hubiera convenido usar tuberías sin nombre: intuitivamente vemos que la implementación sería más compleja porque el proceso DES solo trabaja en un par de tuberías entrada y salida específicas y por tanto para compartirlas debería darse de forma necesaria un intercambio de información entre instancias del conector.

Sin embargo, el precio a pagar hubiera merecido la pena para el extra de seguridad que nos ofrecen las tuberías sin nombre: no se expone ninguna tubería en el sistema de archivos que otros procesos maliciosos externos pudieran leer y escribir.

Es por eso que hubiéramos escogido la vía de las tuberías sin nombre: no obstante, encontramos que en Unix no es posible implementar una solución así sin permisos de administrador.

Esto se debe a que un proceso-conector que quisiera un duplicado/compartición de una tubería sin nombre tendría que acceder a los descriptores de fichero de procesos externos, lo cual no es viable desde unos permisos estándar de usuario —no contemplamos que nuestro conector requiera permisos de administrador: no tiene sentido—. Tendría, en todo caso, que pedir un duplicado al proceso-conector dueño de las tuberías, pero este último no puede hacer nada al respecto, puesto que los descriptores de fichero son simplemente entradas a una tabla específica de cada proceso y, por tanto, de un proceso externo. No existe una función Unix que permita duplicar descriptores de fichero destinándolos a procesos objetivo (en Windows sí, como veremos).

En resumen, usar las tuberías sin nombre en nuestro caso requeriría, como mínimo, permisos de administrador.

La otra alternativa posible sería usar *Unix domain sockets*. Vemos de inmediato que esta alternativa es imposible puesto que no podemos esperar que DES abra por sí solo un *socket* (es un ejecutable ya compilado, no podemos modificarlo). Estas consideraciones con respecto a los *sockets* también aplicarán al caso de Windows y por lo tanto nos abstendremos de mencionarlo allí.

Una vez justificado el uso de tuberías con nombre, recorramos la casuística que acabamos de presentar: para ello, primero tenemos que establecer las primeras condiciones. Recordemos que ya habíamos inicializado la memoria compartida y los cerrojos. Ahora justo antes de llegar a la casuística descrita por el algoritmo 3.1, se tiene que determinar lo siguiente:

1. Si en el directorio de trabajo existen otras conexiones en curso cuyo proceso global DES alude a un ejecutable distinto a nuestro `DES_EXEC`.
2. Si no (o no hay conexiones previas, o las conexiones trabajan con nuestro mismo `DES_EXEC`), comprobar si procede abrir el proceso global DES o por el contrario conectarse al mismo si ya estaba abierto.

En el primer paso simplemente debemos mirar en memoria compartida un valor que alude a la ruta del ejecutable. Si este valor está escrito y alude a un ejecutable distinto a nuestro `DES_EXEC` hay conflicto[9]: se corta nuestra conexión y se informa al usuario que debe cambiar de directorio de trabajo o aludir al mismo ejecutable que es utilizado por las conexiones previas.

En el segundo paso simplemente consultamos en memoria compartida un valor `des_process_created`: este valor se inicia a `false` cuando se crea el fichero de memoria compartida.

Ahora sí se puede determinar entonces en qué caso estamos:

Caso I. *La aplicación llamadora es el único usuario conectado actualmente al directorio de trabajo (i.e., el número de usuarios conectados es cero previamente a la llamada).*

- «*Creamos las tuberías que nos proporcionarán la comunicación bidireccional con un proceso global DES que ahora nos dispondremos a crear.*»

Llamamos a la función POSIX `mkfifo` para las tuberías `driver_to_des_in_wpipe` y `driver_to_des_out_rpipe` por medio de sus respectivos nombres que obtuvimos con el *hashing* (véase el listado 3.3).

```
1 // driver/connect.cc (DESODBC)
2 ...
3 mkfifo(IN_WPIPE_NAME, 0666)
4 ...
```

[9] La causa de por qué esto constituía un conflicto se estudió en la sección 2.2.3.

```
5 mkfifo(OUT_RPIPE_NAME, 0666)
6 ...
```

Listado 3.3: Creación de tuberías en sistemas basados en Unix.

- *«Creamos un proceso DES a partir de la ruta, y se le asocian las tuberías que hemos creado sobre su entrada y salida.»*

La idea será llamar `fork()` para que un proceso hijo gestione lo relativo al futuro proceso DES para que, finalmente, este proceso hijo termine por lanzar con `execlp` sustituyendo su mapa de memoria por el de DES. Veámoslo con detenimiento: supongamos que estamos en el proceso hijo (listado 3.4).

```
1 // driver/connect.cc (DESODBC)
2 pid_t pid = fork();
3 ... if (pid == 0) {
```

Listado 3.4: Llamada a `fork()`.

Veamos el siguiente código del listado 3.5. Desde el proceso hijo, se abren las tuberías `IN_WPIPE_NAME` y `OUT_RPIPE_NAME`, pero como ahora estamos desde la perspectiva de lo que será el proceso global DES, estas tuberías son para leer y escribir respectivamente. Esto será lo contrario a la perspectiva desde el conector (el conector escribe en `in_wpipe` para pasar información a la entrada de DES, por tanto, DES debe leer de esta escritura; y lo análogo con `out_rpipe`).

```
1 // driver/connect.cc (DESODBC)
2 dbc->driver_to_des_in_rpipe = open(IN_WPIPE_NAME, O_RDONLY);
3 ...
4 dbc->driver_to_des_out_wpipe = open(OUT_RPIPE_NAME, O_WRONLY);
```

Listado 3.5: Apertura de tuberías en sistemas basados en Unix desde el proceso hijo.

(nótese que los nombres de los descriptores de fichero ahora se conciben desde la perspectiva de DES —`in_rpipe` y `out_wpipe`—).

Acabamos de abrir los ficheros respectivos a las tuberías con nombre, solo que desde el lado de DES. ¿Cómo asociamos estos descriptores de fichero a la entrada y salida del proceso global DES que estamos por lanzar? Hacemos uso de `dup2` para que estas tuberías con nombre se asocien con entrada y salida actual estándar del proceso hijo en el que estamos. Esto es lo que se realiza en el listado 3.6.

```
// driver/connect.cc (DESODBC)
dup2(driver_to_des_in_rpipe, STDIN_FILENO);
dup2(driver_to_des_out_wpipe, STDOUT_FILENO);
```

Listado 3.6: Asociación de tuberías a la entrada y salida de DES con `dup2`.

A continuación, guardamos el PID del proceso hijo en memoria compartida (listado 3.7), pues este PID será el del proceso DES que procederá a sustituir el mapa de memoria del proceso hijo (necesitamos el PID por si hay que llamar a `kill` llegado el momento).

```
// driver/connect.cc (DESODBC)
shmem->DES_pid = getpid();
```

Listado 3.7: Guardando el PID de DES en memoria compartida.

Finalmente, lanzamos el proceso DES, sustituyendo el mapa de memoria del proceso hijo por el de un proceso DES (listado 3.8).

```
// driver/connect.cc
execlp(des_path, des_path, nullptr);
```

Listado 3.8: Ejecución del proceso DES con `execlp`.

Nótese que con `execlp`, solo hemos cambiado el mapa de memoria del proceso hijo. Esto es lo que permite que tanto la entrada y salida del proceso hijo, así como su PID, permanezcan invariables, y por tanto ahora estén asociadas ahora a DES. Conceptualmente, es como si nuestro proceso hijo simplemente ahora pasara a *comportarse como DES*.

- «*Se consume el mensaje inicial que DES proporciona en su inicio.*»

Esta es tarea del proceso padre, que recordemos, es el que corresponde a la perspectiva del conector. Para ello, debe abrir las tuberías (ahora desde la perspectiva del conector) y almacenar los descriptores de fichero en la estructura entorno (véase el listado 3.9).

```
// driver/connect.cc (DESODBC)
dbc->driver_to_des_in_wpipe = open(IN_WPIPE_NAME, O_WRONLY);
...
dbc->driver_to_des_out_rpipe = open(OUT_RPIPE_NAME, O_RDONLY);
```

Listado 3.9: Apertura de tuberías en sistemas basados en Unix desde el proceso padre.

Posteriormente, procedemos a leer el mensaje inicial de DES llamando a la función POSIX `read` sobre nuestro `dbc->driver_to_des_out_rpipe`: simplemente debemos leer sobre la salida hasta que se consume por completo la tubería de lectura.

Finalmente, almacenamos en memoria compartida `des_process_created = true` y `n_clients = 1`, y acto seguido liberamos el cerrojo asociado a la memoria compartida.

Caso II. *La aplicación llamadora no es el primer usuario conectado actualmente al directorio de trabajo (ya existe un proceso global DES ejecutándose desde ese directorio de trabajo).*

- *«Nos conectamos a las tuberías que ya fueron previamente creadas por el primer usuario.»*

Simplemente debemos:

1. Obtener las tuberías y guardar sus descriptores de fichero en nuestra estructura entorno —exactamente lo que hacía el proceso padre en el caso anterior—.

2. Actualizar en memoria compartida el número de clientes, incrementándolo en uno.

3. Finalmente, liberar el cerrojo de la memoria compartida.

En la sección de desconexión (3.5.2.3), comentaremos cómo todas estas variables IPC llegan a término, tanto si el usuario quiere cerrarlas llamando a las funciones correspondientes, como si la conexión se corta abruptamente por señales del sistema / corte del suministro.

Caso Windows En esta sección nos extenderemos más sobre el aspecto IPC que implementa Windows; estos conocimientos, a diferencia de los de Unix, no son vistos en los actuales planes de estudio de informática.

Tras contemplar la sencillez conceptual de la solución que nos ofrece Unix para un problema como el nuestro, la idea es hasta qué punto lo podemos replicar en Windows. Emprendamos todo el proceso en orden.

La primera diferencia es que los descriptores de fichero de Unix (`int`) se traducen a manejadores (`HANDLE`), que son punteros hacia un área de memoria, en línea con los conceptos ya vistos. En la tabla 3.1 no es necesario comentar la correspondencia entre memoria compartida y cerrojos entre sistemas Unix y Windows, pues en este caso es directa.

El aspecto IPC que definirá una gran ruptura entre la arquitectura propuesta para la conexión en Unix y la de Windows es el de las tuberías.

Técnicamente, Windows proporciona una solución para tuberías con nombre [20], estableciendo una comunicación bidireccional entre lo que llaman *pipe server* y los *pipe clients*.

Las funciones del *pipe server* son:

1. Crear una tubería con nombre (`CreateNamedPipe` —`winbase.h`—)

2. Esperar a que los procesos clientes se conecten a una instancia de una tubería con nombre (`ConnectNamedPipe` —`namedpipeapi.h`—).

La función primordial del *pipe client* por otro lado es conectarse a una tubería con nombre ya creada, por medio de `CreateFile` (`fileapi.h`) o `CallNamedPipe` (`winbase.h`).

La pregunta que cabe hacerse es: ¿quién debe ser el *pipe server*? ¿Quién reúne las características para hacerlo? DES en sí no tiene la posibilidad de hacer de servidor: es un ejecutable ya compilado y que no procede modificar.

Si DES no puede ser un *pipe server*, deberá entonces serlo alguna conexión, y desde luego, siempre deberá serlo la primera, por lo menos inicialmente —si no, nunca se podría conectar—. Pero, ¿qué ocurre si el primer usuario se desconecta, y quedan otros clientes en activo? Las opciones son:

a) Que venga predefinido directamente que todas las conexiones hagan de servidor.

b) Pasar el «testigo» del rol del servidor a otro usuario.

c) Un ejecutable externo global hace de *pipe server*.

La opción a) es problemática puesto que si bien Windows permite que múltiples servidores coexistan, desde el servidor se obtiene un *server-end handle* único de la tubería con nombre, y por tanto no compartido entre servidores. Esto es un problema porque a DES solo se le puede asignar unos manejadores determinados de entrada y salida, y cambiárselos en tiempo de ejecución de DES no es una opción pues requeriría como mínimo permisos de administrador (como ahora veremos).

La opción b) también es problemática porque el usuario puede haber finalizado abruptamente el programa. Por tanto, el *pipe server* debería enviar *server-end handles* duplicados conforme se vayan conectando nuevos usuarios, para que estos los tengan a su disposición si se desconectara el *pipe server*. Pero esto supone la implementación de un sistema de intercambio de mensajes demasiado complejo: es una posibilidad, pero a continuación veremos que las tuberías sin nombre ofrecen una solución algo más sencilla, pero que sobre todo, como ya hemos adelantado para Unix, es más segura al no exponer una tubería con nombre a la vista de otros procesos potencialmente maliciosos.

La opción c) listada antes no es viable: el conector ODBC debe constituir un único archivo `.dll`.

Vamos a descartar el uso de tuberías con nombre por:

- Implementación compleja.
- Falta de seguridad.
- Existe la posibilidad de que las tuberías con nombre sean bloqueadas por el firewall de Windows[10].
- Las tuberías sin nombre son más seguras y su implementación es algo más sencilla (y posible sin permisos de administrador, como ahora veremos).

Observemos el resto de soluciones posibles:

a) Utilizar posibles manejadores entrada y salida expuestos por parte del proceso DES.

b) Utilizar tuberías sin nombre.

A priori, la opción a) es una posibilidad. Podemos observar los manejadores de DES por medio del software *Process Explorer v17.06*[11] y ver que, en principio, existen manejadores de entrada y salida expuestos (véase la figura 3.8).

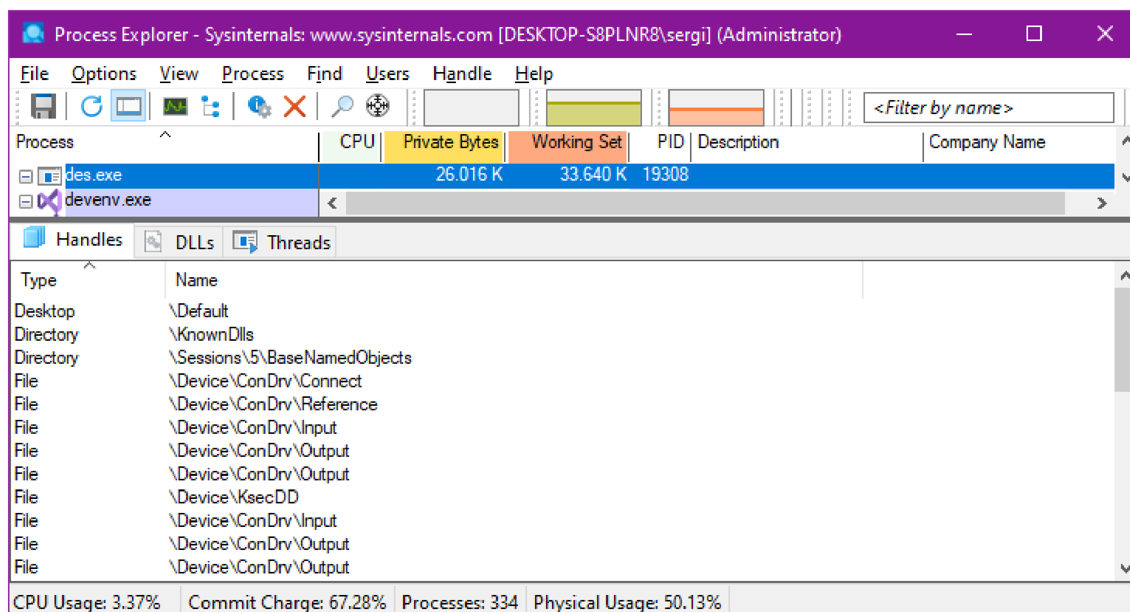


Figura 3.8: Vista de los manejadores que DES expone, en el software *Process Explorer*.

[10] Son varias las entradas en foros donde usuarios denuncian cómo el firewall de Windows bloquea tuberías con nombre en algunos contextos, principalmente por las características de la máquina del usuario —es decir, algo que desde el conector no podemos controlar—. Véase, por ejemplo: `https://superuser.com/questions/462443/what-are-reasons-for-local-windows-named-pipes-to-fail` (recuperado el 16 de abril de 2025).

[11] `https://learn.microsoft.com/en-us/sysinternals/downloads/process-explorer`

Sin embargo, para haber accedido a los manejadores, ya advertimos en la figura 3.8 en el título de la ventana que ha hecho falta ejecutar *Process Explorer* en modo administrador. En efecto, se requieren ciertos permisos otorgados a los administradores para abrir manejadores de procesos externos y poseer un acceso completo a los mismos[12]; además, deseamos que el sistema debería ser opaco para el conector, con único punto expuesto los archivos específicos de memoria compartida y cerrojos.

Solo nos queda la opción b). ¿Es posible que múltiples procesos conector accedan a tuberías sin nombre asociadas a DES? Parece que esa es precisamente la situación propicia para las tuberías con nombre y por tanto, que las tuberías sin nombre no pueden ser compartidas entre múltiples procesos. Sin embargo, hemos diseñado una arquitectura que permite solucionar todos los obstáculos encontrados a partir de tuberías sin nombre.

La idea central es que los nuevos clientes que se conecten pidan a otro cliente conectado que le genere un duplicado de los manejadores. Para ello, el cliente requerido para copiar llama a `DuplicateHandle` con el PID del proceso objetivo que usará la copia (esto es necesario pues la copia es específica y exclusiva para cada proceso, no existen copias «globales»). Existe por tanto un intercambio de información entre el proceso que pide la copia y el proceso que la genera, dentro únicamente de los confines de la memoria compartida.

Este intercambio de información en el proceso de copia motiva el uso de objetos de evento [22], que son objetos de sincronización que envían/reciben señales, indicando que cierto evento ha tenido lugar. Distinguimos tres eventos: «pedida de copia del manejador» (`request_handle_event`), «copia del manejador enviado» (`handle_sent_event`) y «petición de cierre de conexión» (`finishing_event`). Veremos en qué contextos se usan cada uno de estos eventos.

Estamos en posición de empezar el recorrido presentado en el algoritmo 3.1. Los antecedentes son que hemos podido inicializar la memoria compartida y los cerrojos. Ahora comprobamos los preliminares (existencia de otros clientes conectados y posibles conflictos) tal y como hicimos en Unix, y acto seguido, podemos ya empezar con la casuística:

Caso I. *La aplicación llamadora es el único usuario conectado actualmente al directorio de trabajo (i.e., el número de usuarios conectados es cero previamente a la llamada).*

- *«Creamos las tuberías que nos proporcionarán la comunicación bidireccional con un proceso global DES que ahora nos dispondremos a crear.»*

Creamos tuberías sin nombre para entrada y salida (veáse el listado 3.10).

[12] «*To open a handle to another process and obtain full access rights, you must enable the SeDebugPrivilege privilege.*» [21].

```
// driver/connect.cc (DESODBC)
...
...CreatePipe(&dbc->driver_to_des_out_rpipe, &dbc->driver_to_des_out_wpipe,
                &des_pipe_sec_attr,
                des_pipe_buf_size)
...
...CreatePipe(&dbc->driver_to_des_in_rpipe, &dbc->driver_to_des_in_wpipe,
                &des_pipe_sec_attr, des_pipe_buf_size)
```

Listado 3.10: Creación de tuberías sin nombre en Windows.

- *«Creamos un proceso DES a partir de la ruta, y se le asocian las tuberías que hemos creado sobre su entrada y salida.»*

En Windows podemos crear un proceso por medio de la función `CreateProcess` (`processthreadsapi.h`). La funcionalidad clave que usaremos es que en los parámetros de esta función podemos enviar una estructura `STARTUPINFO`, y en cuyo interior podemos establecer los manejadores de entrada y salida que estarán asociados con el proceso a ejecutar. Nos limitamos entonces aquí a asignarles los extremos de las tuberías que ya hemos inicializado previamente. Este proceso completo puede observarse en el listado 3.11.

```
// driver/connect.cc (DESODBC)
...
ZeroMemory(&dbc->startup_info_unicode, sizeof(dbc->startup_info_unicode));
...
env->startup_info_unicode.hStdError = dbc->driver_to_des_out_wpipe;
env->startup_info_unicode.hStdOutput = dbc->driver_to_des_out_wpipe;
env->startup_info_unicode.hStdInput = dbc->driver_to_des_in_rpipe;
...
... CreateProcessW(NULL, des_path, NULL, NULL, TRUE, DETACHED_PROCESS |
                   CREATE_NEW_PROCESS_GROUP, NULL, des_working_dir,
                   &dbc->startup_info_unicode,
                   &dbc->process_info)
...
//Almacenamos el PID del proceso global DES
dbc->shmem->DES_pid = dbc->process_info.dwProcessId;
```

Listado 3.11: Creación del proceso DES en Windows.

- *«Se consume el mensaje inicial que DES proporciona en su inicio.»*

Simplemente se llama a la función `ReadFile` (`fileapi.h`) con el manejador `env->driver_to_des_out_rpipe` conservando el mismo proceso de lectura del mensaje inicial que hemos descrito en el caso Unix. Para concluir todo el proceso, se hace lo siguiente:

1. Se pone `des_process_created`, de la memoria compartida, a `true`.
2. Se almacena el PID asociado a la instancia del conector en la lista de conexiones activas de memoria compartida.
3. Se crea y ejecuta un hilo que se ejecuta en segundo plano, asociado a la duplicación y compartición de manejadores.
4. Se libera el cerrojo de la memoria compartida.

En el siguiente caso tratamos cómo se duplican y comparten los manejadores.

Caso II. *La aplicación llamadora no es el primer usuario conectado actualmente al directorio de trabajo (ya existe un proceso global DES ejecutándose desde ese directorio de trabajo).*

- *«Nos conectamos a las tuberías que ya fueron previamente creadas por el primer usuario.»*

La terminología que usaremos es proceso *petitioner* para el proceso que pide que se le dupliquen manejadores, y proceso *petitionee* para que el proceso encargado de duplicar y compartir los manejadores.

Nos encontramos en la perspectiva del proceso *petitioner*. Adquiere los cerrojos de memoria compartida y `request_handle_mutex`, y acto seguido hace lo siguiente:

1. Pone en memoria compartida, en un `pid_handle_petitioner`, su PID.
2. Procede a decidir quién será el *petitionee*. Para ello, toma un PID aleatorio de la lista de PIDs activos (si este no responde, vuelve a intentar con otro aleatorio), y lo coloca en una variable compartida `pid_handle_petitionee`. Libera el cerrojo de la memoria compartida, envía la señal de `request_handle_event`, y procede a esperar al evento `handle_sent_event`, pues ahí será cuando los manejadores duplicados estén listos para él.

Saltamos a la perspectiva del *petitionee*. El *petitionee*, previo a todo esto, ejecutaba un hilo en el que esperaba periódicamente que le llamaran. Recibe una señal `request_handle_event` a causa del *petitioner*, y examina en memoria compartida si `pid_handle_petitionee` corresponde a su propio PID: si no, continúa esperando a futuros eventos `request_handle_event`; si resulta ser él el *petitionee* demandado, procede a duplicar los manejadores de forma específica para el *petitionee* (con su PID) y colocarlos en memoria compartida. Véase el listado 3.12.

```
1  // driver/connect.cc (DESODBC)
2
3  HANDLE petitioner_process_handle =
4          OpenProcess(PROCESS_DUP_HANDLE, TRUE,
5                      shmem->handle_sharing_info.pid_handle_petitioner);
6
7  ... DuplicateHandle(GetCurrentProcess(), dbc->driver_to_des_out_rpipe,
```

```
8                       petitioner_process_handle,
9                       &shmem->handle_sharing_info.out_handle, 0, TRUE,
10                      DUPLICATE_SAME_ACCESS);
11
12 ... DuplicateHandle(GetCurrentProcess(), dbc->driver_to_des_in_wpipe,
13                      petitioner_process_handle,
14                      &shmem->handle_sharing_info.in_handle, 0, TRUE,
15                      DUPLICATE_SAME_ACCESS);
```

Listado 3.12: Duplicado de manejadores de tubería desde el proceso *petitionee* hacia el proceso *petitioner*.

Acto seguido, activa el evento `handle_sent_event` para que el *petitioner* sepa que puede recoger sus nuevos manejadores.

Volvemos entonces a la perspectiva del *petitioner*, que ahora procede a almacenar en su estructura `ENV` los manejadores de la memoria compartida, limpiar la estructura de un tipo `HandleSharingInfo` para futuras comparticiones, y liberar los cerrojos. Este proceso puede verse en el listado 3.13.

```
1  // driver/connect.cc (DESODBC)
2  ...
3  dbc->driver_to_des_out_rpipe = dbc->shmem->handle_sharing_info.out_handle;
4  dbc->driver_to_des_in_wpipe = dbc->shmem->handle_sharing_info.in_handle;
5
6  // we reset the structure once we have saved the handles
7  dbc->shmem->handle_sharing_info.in_handle = NULL;
8  dbc->shmem->handle_sharing_info.out_handle = NULL;
9  dbc->shmem->handle_sharing_info.pid_handle_petitionee = 0;
10 dbc->shmem->handle_sharing_info.pid_handle_petitioner = 0;
11 ...
```

Listado 3.13: Recepción de manejadores de tubería y limpieza de la estructura `HandleSharingInfo` de memoria compartida, por parte del *petitioner*.

3.5.2.3. Desconexión

La forma estándar y correcta de desconectarnos es llamando a `SQLDisconnect`. Sin embargo, un código robusto y seguro también debe cubrir los casos donde las aplicaciones se cierran en otras circunstancias. Veremos las dos posibilidades.

`SQLDisconnect` La función `SQLDisconnect` toma como argumento el manejador de conexión, y su objetivo es, en abstracto, cerrar la conexión [23]. Nuestra interpretación será que tras esta llamada el manejador sigue apuntando a nuestra estructura `DBC`, solo que esta ya no tiene punteros válidos a los recursos IPC. Para destruir la estructura `DBC` hay que llamar a la función `SQLFreeHandle` con el manejador de conexión en cuestión.

En esta función, si procede, nos encargaremos de liberar la memoria compartida, cerrojos, tuberías, y posiblemente el proceso DES. Es por esto que constituye un cierre seguro de la conexión, y es estándar en ODBC llamar a esta función para desconectarnos. Es importante señalar antes de nada lo siguiente: tanto en Windows como en Unix, «cerrar» un manejador/descriptor de fichero se traduce en «liberar un usuario de dicho archivo», de modo que esos recursos aún existen para los otros clientes conectados. Sin embargo:

1. En Windows, al cerrar un manejador (`CloseHandle` —`handleapi.h`—), no se destruye el objeto en el sistema al que alude, pero sí cuando pase a no existir ningún manejador apuntándolo [24].

2. En Unix, al cerrar un descriptor de fichero (`close`), no se destruye el objeto en el sistema al que alude, ni siquiera si no existe ningún descriptor de fichero abierto. Para destruirlo debemos de llamar a alguna función como `unlink`.

Más tarde analizaremos en detalle este comportamiento y las consecuencias que implica cuando se da un cierre no seguro.

En `SQLDisconnect`, antes de desconectarnos, examinamos en memoria compartida cuántos clientes hay conectados. Distinguimos casos:

a) Existen más clientes

Entonces en base a nuestro análisis simplemente debemos llamar a los *close* de los recursos sin más complicación. Merece la pena detenerse en el caso de Windows, puesto que tenemos que cerrar manualmente nuestro hilo de compartición de tuberías, ejecutado en segundo plano perpetuamente, si no queremos que la aplicación se cuelgue. En Windows, el procedimiento completo sería:

1. Se pone en memoria compartida el PID actual como *petitioner*.

2. Se envía una señal «cerrando» (el evento `finishing_event` que listamos anteriormente).

3. En cualquier cliente, el hilo que comparte las tuberías captura esta señal, y examina en memoria compartida el PID del *petitioner*; si coincide con su PID, «se da por aludido» y se sale del bucle infinito de espera, pudiendo cerrarse el hilo.

4. Se procede a cerrar los manejadores de tubería. Recordemos que mientras exista algún otro manejador apuntando a las tuberías, Windows no eliminará las tuberías.

5. El cliente elimina de memoria compartida su información como cliente: ahora no aparecerá disponible al resto de clientes.

6. Cierra sus manejadores a la memoria compartida y cerrojos.

b) Somos el último cliente restante

Cuando se desconecta el último usuario del directorio de trabajo, tenemos primero que

- Terminar el proceso DES. Para ello, se envía el comando DES `/q`, pues además de ser un cierre seguro, es posible que no podamos llamar a `kill` (Unix) o `TerminateProcess` (Windows, `processthreadsapi.h`) por una cuestión de permisos.
- Cerrar los manejador/descriptores de fichero.
    - En el caso de Unix, además destruir estos recursos con `unlink`.

Veamos a ver qué ocurre si se cierra el conector sin haber llamado a `SQLDisconnect`.

Cierre «abrupto» del conector. Windows y Unix Cuando una aplicación que ha alojado un manejador de conexión se termina abruptamente sin llamar a `SQLDisconnect` ni liberar el manejador de conexión, el ODBC Driver Manager tampoco realiza estos pasos [25; 26]: recordemos que el ODBC Driver Manager no es un proceso separado, sino una biblioteca cargada en el espacio de memoria de la aplicación.

Lo que ha ocurrido es que al finalizar la aplicación de golpe, simplemente se ha destruido todo su espacio de memoria: el ODBC Driver Manager, el conector, la memoria a la que apuntan los manejadores... pero todo ello, sin haber ejecutado ninguna rutina que nos permitiera haber realizado un «cierre seguro».

Entonces, sabemos que los recursos ODBC se destruyen junto con la aplicación. Pero nos debemos preguntar ¿qué ocurre con los recursos relacionados puramente con DES —tuberías, nuestra memoria compartida, cerrojos, y proceso DES—? No podemos permitir que nuestros recursos relacionados con DES queden desatendidos y desperdigados por el sistema. No podemos contar con que la aplicación siempre vaya a cerrar apropiadamente, ya sea por mala programación o por término inesperado del proceso de la aplicación.

Windows nos ayuda en esta tarea: por defecto, destruye los archivos de memoria compartida y tuberías cuando ningún proceso está accediendo a ellos[13].

El caso de qué ocurre con el proceso DES es especial. A priori, parecería lógico que Windows no tuviera razones para terminarlo él mismo. En efecto, se puede comprobar con una aplicación de prueba que si llama a un simple `CreateProcess` (`processthreadsapi.h`) y acto seguido la aplicación termina, el proceso llamado sigue ejecutándose.

[13] En [27] se explica que existe un parámetro `OBJ_PERMANENT` para forzar a que los objetos no se autodestruyan y que, por defecto, «*such objects are deleted when all open handles to them are closed*».

Sin embargo, en nuestro conector, previamente a la llamada de `CreateProcess` (`processthreadsapi.h`) establecimos en su estructura `STARTUPINFO` que DES se inicialice con una entrada y salida asociadas a unas tuberías. Parece que cuando las tuberías dejan de existir, el proceso procede a finalizarse. No hemos encontrado documentación que avale este comportamiento, pero si se hace el experimento se puede comprobar que si se usa `CreateProcess` (`processthreadsapi.h`) tanto enlazando tuberías como no enlazándolas, *ceteris paribus* en un caso el proceso finaliza cuando la aplicación termina y en el otro no.

En resumen, hemos comprobado empíricamente[14] que Windows se encarga de liberar los recursos por nosotros. La explicación al respecto es que creemos que, tras finalizarse una aplicación abruptamente cuando esta constituye la única conexión a un proceso global DES, ocurre lo siguiente:

1. Windows detecta que no hay ningún manejador abierto asociado a la memoria compartida y las tuberías.
2. Por tanto, Windows procede a destruir estos objetos.
3. El proceso DES se encuentra con que sus `stdin`/`stdout` corresponden a tuberías destruidas.
4. El proceso DES procede en consecuencia a terminarse.

Sea esta explicación precisa o no, en cualquier caso no tenemos que hacer ninguna modificación adicional, puesto que sabemos con certeza que Windows libera los recursos adecuadamente. El caso extremo en el que hay un corte del suministro eléctrico no supone ningún problema: como todas las instancias del conector se ejecutan en la misma máquina, se destruirán todas las conexiones por igual. Apuntamos como nota final que en Windows si se cierra una aplicación sin llamar a `SQLDisconnect`, pese a que no se haya podido eliminar el cliente a sí mismo de la lista en memoria compartida (tal y como describimos en la implementación de `SQLDisconnect`), cuando otro cliente le pida que comparta sus tuberías obtendrá un *timeout*, y acto seguido procederá a eliminar a este cliente fantasma de la lista.

Por desgracia, Unix no nos facilitará el cierre seguro. Si se cierra de forma no segura una aplicación que conforme una única conexión restante a cierto proceso DES:

- No se destruye el archivo de memoria compartida, creado con `shmget`, de la RAM.
- Las tuberías permanecen en disco, y en el caso de macOS, ocurre igual además con los cerrojos[15].

[14] Entre otras utilidades, hemos utilizado AccessChk para ello. En la sección de *software* empleado (3.8) trataremos esto más a fondo.

[15] Recordemos lo comentado para macOS: al no implementar la función `sem_init`, debe llamar a semáforos con nombre, almacenados en disco. En GNU/Linux, sin embargo, introducimos los semáforos (sin nombre) en la propia memoria compartida.

- El proceso DES seguirá activo (tras haber hecho `fork()` y destruirse el padre, el proceso hijo huérfano —DES— es adoptado por el proceso superior al padre en la jerarquía, y siempre existirá uno —`init` o alguno de sus hijos—).

Por fortuna, estos escenarios son raros: ocurrirían cuando hay un corte del suministro eléctrico o bien se envía `SIGKILL` a la aplicación. Cuando se envía `SIGKILL` se destruye el proceso de forma inminente: el sistema no se permite capturar esta señal. Sin embargo, un usuario no debe recurrir al envío de `SIGKILL` salvo que sea estrictamente necesario. En efecto, lo correcto en la mayoría de situaciones es enviar señales `SIGINT` o `SIGTERM` para cierre seguro de la aplicación, permitiendo a estas cerrar ficheros y conexiones antes de destruir el proceso: esto es porque estas señales, a diferencia de `SIGKILL`, pueden ser capturadas y gestionadas.

La solución a priori sería que nuestro conector interceptara estas señales y las gestionara. Esto conlleva el problema de que puede interferir con la aplicación usuario en caso de que esta también intercepte estas señales. Optamos por añadir en nuestro conector la directiva `atexit`, que permite llamar una función específica en caso de que la aplicación no cierre de forma inesperada, y procede después a la destrucción de la biblioteca cargada [28]. Véase el listado 3.14.

```
1  //driver/connect.cc (DESODBC)
2
3  void safe_close_connections() {
4    for (auto hdbc : active_dbcs_global_var) {
5      hdbc->close();
6    }
7  }
8
9  __attribute__((constructor))
10 void init() {
11   atexit(safe_close_connections);
12 }
```

Listado 3.14: Cierre seguro forzado con `atexit`.

Este código tiene el mismo efecto que llamar a `SQLDisconnect` a todos los manejadores de conexión activos.

No obstante, en el caso en el que se haya cerrado el conector de forma no segura y se pretendan abrir recursos ya creados en conexiones pasadas, se lanzará un error con mensaje al usuario dictándole que elimine manualmente los recursos pertinentes.

3.5.3. Funciones de manejo de sentencias

En el manejo de sentencias es donde se despliega la funcionalidad primordial de un conector ODBC: gestionar las sentencias siendo intermediario entre la aplicación y el DBMS destino. Podemos clasificar en este ámbito las siguientes funciones API: `SQLBindParameter`, `SQLCancel`, `SQLCloseCursor`, `SQLDescribeParam`,

SQLExecute, SQLExecDirect, SQLFreeStmt, SQLGetCursorName, SQLNumParams, SQLParamData, SQLParamOptions, SQLPrepare, SQLPutData y SQLSetCursorName.

Estas funciones sirven para preparar y enviar las sentencias; las funciones que lidian con los resultados obtenidos se estudiarán en el apartado 3.5.4.

Por fortuna, podremos reutilizar gran parte del código relativo a estas funciones de manejo de sentencia. MyODBC proporciona como clase principal STMT: un puntero a un objeto de esta clase constituye un manejador de sentencia. En esta clase existe lo esperado, como por ejemplo, listas de los distintos manejadores de descriptor, los atributos y parámetros enlazados, la cadena que contiene la consulta en sí, etcétera.

Nuestra tarea es reutilizar lo máximo de esta clase y eliminar los elementos relativos estrictamente a MySQL. Observamos que durante el proceso de consulta dentro de las funciones API encargadas de la ejecución, MyODBC solo entra en contacto con MySQL en el momento exacto de enviarle la consulta, que constituye una única función auxiliar de biblioteca externa que se limita a enviar a MySQL Server la consulta. En el listado 3.15 se puede observar un punto del código del conector donde está por lanzarse una consulta.

```
1 // driver/execute.cc (MyODBC)
2    ...
3 154: if (!get_result_metadata(stmt, FALSE))
4    ...
```

Listado 3.15: Ejemplo de lugar en MyODBC donde se lanza la consulta a MySQL Server.

Esta función get_result_metadata hace una llamada a una biblioteca externa asociada a MySQL, quien recogiendo ciertos atributos del objeto sentencia procede a comunicarse con la base de datos en nuestro sistema. La función externa concreta a la que se llama es mysql_store_result, cuya cabecera puede observarse en el listado 3.16.

```
1 // include/mysql-8.0/mysql.h (MyODBC)
2 468: MYSQL_RES *STDCALL mysql_store_result(MYSQL *mysql);
```

Listado 3.16: Cabecera de la función mysql_store_result en MyODBC.

Vemos que lo único que nos interesa es que devuelve un puntero estructura MYSQL_RES. En el listado 3.17 podemos observar sus atributos.

```
1 // mysql.h (MyODBC)
2 typedef struct MYSQL_RES {
3   uint64_t row_count;
4   MYSQL_FIELD *fields;
```

```
5     struct MYSQL_DATA *data;
6     MYSQL_ROWS *data_cursor;
7     unsigned long *lengths; /* column lengths of current row */
8     MYSQL *handle; /* for unbuffered reads */
9     const struct MYSQL_METHODS *methods;
10    MYSQL_ROW row; /* If unbuffered read */
11    MYSQL_ROW current_row; /* buffer to current row */
12    struct MEM_ROOT *field_alloc;
13    unsigned int field_count, current_field;
14    bool eof; /* Used by mysql_fetch_row */
15    /* mysql_stmt_close() had to cancel this result */
16    bool unbuffered_fetch_cancelled;
17    enum enum_resultset_metadata metadata;
18    void *extension;
19  } MYSQL_RES;
```

Listado 3.17: Estructura `MYSQL_RES` en MySQL.

Observamos que, rellenando apropiadamente esta estructura, nuestro conector no necesita más para conocer el resultado de la consulta. Por tanto, nuestra tarea es reutilizar de `MYSQL_RES` el máximo número de atributos, eliminar los que sean superfluos, y rellenarlos apropiadamente con los valores fruto de las consultas.

Estudiando el código fuente de MySQL Server, que es donde se implementan todas estas funciones de biblioteca externa, es como hemos podido conocer cómo se espera rellenar los atributos y así garantizar mismas postcondiciones para las mismas precondiciones.

La parte relacionada a los «resultados en bruto» corresponde al atributo `struct MEM_ROOT *field_alloc`. La idea central será guardar los resultados que DES arroje en una estructura interna análoga (`ResultTable* internal_table`) tras haber extraído la información de los resultados de la TAPI obtenidos con una lectura de la salida de DES. El resto de atributos también son rellenados convenientemente, en base a estos resultados obtenidos.

En el listado 3.18 puede observarse cómo se reescribiría para DES el proceso de ejecución de una consulta.

```
1   // driver/execute.cc (DESODBC)
2   ...
3   full_query = "/tapi " + query + '\n';
4   ...WaitForSingleObject(env->query_mutex, MUTEX_TIMEOUT);
5   ...WriteFile(stmt->dbc->driver_to_des_in_wpipe, full_query_arr,
6                   strlen(full_query_arr),
7                   &bytes_written, NULL)
8   ...ReadFile(stmt->dbc->driver_to_des_out_rpipe, buffer,
9                             BUFFER_SIZE - sizeof(CHAR), &bytes_read, NULL);
10  ...ReleaseMutex(env->query_mutex);
11  ...stmt->last_output = tapi_output;
```

```
12 ...if (!get_result_metadata(stmt, FALSE)) {
13 ...
```

Listado 3.18: Proceso completo de ejecución de consulta en DESODBC.

(código presentado con fines expositivos: el código real está organizado de manera distinta)

Es decir:

1. Construimos la consulta completa: `/tapi` + consulta del usuario, para obtener los resultados en modo TAPI.
2. Adquirir el cerrojo de consulta.
3. Enviar a DES la consulta.
4. Leer la salida que arroja DES.
5. Liberar el cerrojo de consulta.
6. Almacenar esta salida dentro de la estructura sentencia, y llamar a la función `get_result_metadata()` (que ahora ya no incluye del envío de la consulta a MySQL) para construir el resultado final.

Por tanto, vamos a tener que modificar la función `get_result_metadata()` que acabamos de estudiar, que ahora en lugar de llamar a `MYSQL_RES* mysql_store_result()`, llamará a una función `DES_RESULT* des_store_result()`. Esta última función es la que tenemos que escribir desde cero para llegar a las mismas postcondiciones de una llamada a `mysql_store_result()`. Nos extenderíamos demasiado si describiéramos el proceso completo de construcción de la tabla interna en este documento, cuyo único objeto es presentar el funcionamiento general del conector.

Hemos conseguido sustituir el papel de MySQL por el de DES al tratar una consulta. Sin embargo, aún tenemos que hacer frente al manejo de los resultados en sí, que veremos en la próxima sección. En MyODBC, cuando se manejan resultados se hacen llamadas a funciones de biblioteca externa que manipulan la estructura `MYSQL_RESULT`. Nuevamente volveremos a sustituirlas por funciones homólogas de mismo comportamiento.

3.5.4. Funciones de manejo de resultados

Una vez se ejecuta una consulta con las funciones de la API pertinentes, las aplicaciones leen y gestionan los resultados por medio de unas determinadas funciones, que clasificaremos bajo la etiqueta de «manejo de resultados», y que corresponden a `SQLBindCol`, `SQLBulkOperations`, `SQLColAttribute`, `SQLColAttributes`, `SQLDescribeCol`, `SQLExtendedFetch`, `SQLFetch`, `SQLFetchScroll`, `SQLGetData`, `SQLMoreResults`, `SQLNumResultCols`, `SQLRowCount`, `SQLSetPos` y `SQLSetScrollOptions`.

Vamos a prescindir de comentar gran parte de estas funciones, pues han podido ser reutilizadas por completo de MyODBC. Vamos a tratar algunos aspectos importantes que hemos modificado de estas funciones, y para ello recorreremos las funcionalidades distinguiendo sobre dos modos de proceder relativos al manejo de resultados: lectura y modificación.

3.5.4.1. Lectura

Algunas de las funciones citadas tienen como objetivo funcional el de «lectura»; es decir, obtener datos sobre el conjunto de resultados para mostrarlos al usuario.

Cursores Introducimos de antemano el concepto de cursor. El cursor es un concepto que representa en abstracto un puntero que apunta a una fila en el conjunto de resultados, como si de un cursor de texto en ofimática se tratara [12, p. 452]. Las funciones de manejo de resultados harán uso del cursor para realizar sus gestiones. El estándar ODBC distingue los siguientes tipos de cursores:

- *Forward-only*: solo se pueden mover hacia adelante en el conjunto de resultados. Si se quisiera acceder a una fila anterior, habría que reiniciar el cursor [12, p. 452].
- *Scrollable*: permiten acceso a cualquier fila del conjunto de resultados sin importar dirección [12, p. 451].
  - *Static*: con este cursor «*el conjunto de resultados aparenta ser estático*»[16] [12, p. 542]; es decir, los cambios que van realizando otras conexiones no alteran el conjunto de resultados que percibe el cursor.
  - *Dynamic*: son lo contrario de los estáticos; los cambios que realizan otras conexiones sobre el conjunto de resultados pasan a actualizarse sobre la visión del cursor [12, p. 543]. Podemos anticipar que son muy costosos de implementar.
  - *Keyset-driven*: son dinámicos en tanto que perciben las actualizaciones de los valores del conjunto de resultados, pero a su vez mantienen la estructura estática en base a un sistema de claves (*keys*) de la tabla de resultados. Véase [12, p. 543].
- *Block*: más que una categoría de cursor separada de los anteriores, es un atributo del cursor: si apuntan a un bloque de filas en el conjunto de resultados. Permiten por lo tanto leer de una sola tacada un bloque de filas, lo que supone una mayor eficiencia en determinados contextos [29; 30].

No es imprescindible implementar todos los cursores. Por ejemplo, MyODBC ofrece soporte para cursores *forward-only*, *static*, *dynamic* y posibilidad de apuntar en bloque, pero por tanto, no implementa cursores *keyset-driven*.

[16] Traducciones propias al castellano.

DESODBC por su parte, solo implementará de estos anteriores los cursores puramente estáticos: es decir, *forward-only*, *static* y posibilidad de *block*. Este tipo de cursores permiten reutilizar gran parte el código de MyODBC relativo a los mismos, pues al no ser dinámicos, no llaman a funciones de biblioteca externa de MySQL.

Permitir en nuestra implementación algún grado de dinamismo en cursores excedería los objetivos de un trabajo de fin de grado, pues si bien podría llevarse a cabo con los conocimientos del plan de estudios actual, el tiempo a invertir para esta funcionalidad particular se escaparía de lo razonable. De hecho, no es obligatorio implementarlos y algunos conectores ODBC[17] optan por no hacerlo, por lo que pese a no incluirlos, seguiremos cumpliendo con nuestros objetivos mínimos propuestos. Lo incluiremos como posibilidad a explorar en trabajos futuros. Por el momento tenemos que eliminar todo el código de MyODBC con respecto a los cursores de tipo dinámico.

Ahora que conocemos el concepto de cursor, las funciones relacionadas con la lectura se basan en los movimientos de este cursor y devolución de resultados sobre los que se encuentre. Como antes, la filosofía será reutilizar todo el código de MyODBC, únicamente reimplementando funciones que sustituyan a las funciones de biblioteca externa de MySQL. Veamos un ejemplo ilustrativo a continuación.

La función `SQLFetch` se encarga de posicionar el cursor en la siguiente fila / conjunto de filas del resultado que se haya obtenido en la última sentencia. En la implementación de MyODBC, se manipula la estructura `STMT` convenientemente, proceso que replicaremos por completo, y en un momento dado se llama a la función de biblioteca externa `MYSQL_ROW STDCALL mysql_fetch_row(MYSQL_RES *result)`. Nuestra tarea consistió nuevamente en sustituir esta función por una función homóloga que verificara las mismas postcondiciones para las mismas precondiciones, ayudándonos del código fuente de MySQL Server, que incluye la implementación de esta función de biblioteca externa.

Nota sobre los marcadores ODBC ofrece la posibilidad de trabajar por marcadores (*bookmarks*). Un marcador se define simplemente como un valor que identifica una fila del conjunto de resultados. Es decir, el concepto análogo que el propio nombre *bookmark* indica sería un marcapáginas sobre una página de un libro, solo que en el caso de ODBC, se trataría de almacenar «un número de página», recordarlo, y leerlo para que el cursor se posicione sobre «la página» (la fila) llegado el momento [32].

Así, la aplicación puede pedir que el conector coloque un marcador en la fila en la que el cursor se encuentra sobre el conjunto de resultados, y tiempo después pide al conector que ejecute cierta operación sobre la fila recordada: actualizarla, eliminarla, posicionarse sobre la misma, etcétera. (Nótese que la aplicación solo está

[17]Por ejemplo, los conectores de SQLite y Firebird no implementan ningún grado de dinamismo (descartando por tanto cursores dinámicos y *keyset-driven*. En el primer caso lo sabemos examinando el código fuente, en el segundo leyendo su documentación: [31, sección 5.4].

pidiendo trabajar por marcadores: el valor y el significado del marcador en sí solo son conocidos por el conector).

Los marcadores son considerados una funcionalidad avanzada para un conector ODBC[18]. Introducimos este concepto porque luego será necesario pasar sobre el mismo, y en este punto comentar simplemente que DESODBC también lo implementa, puesto que el uso de marcadores no echa mano de funciones de biblioteca externa MySQL que no hayamos reemplazado de antemano.

3.5.4.2. Modificación

Una vez tenemos un conjunto de resultados fruto de haber ejecutado una consulta, no solo tenemos la posibilidad de leer sobre este conjunto, sino también de modificarlo y enviar las modificaciones al DBMS. Las funciones API donde se pueden realizar estas operaciones son `SQLSetPos` y `SQLBulkOperations`.

`SQLSetPos` `SQLSetPos` admite como argumento un número de fila y unas determinadas operaciones a realizar sobre la misma: `SQL_POSITION`, `SQL_REFRESH`, `SQL_UPDATE` y `SQL_DELETE`. Son estas tres últimas las que pertenecen al ámbito «modificación». En versiones anteriores se admitía también `SQL_ADD` para la inserción de filas, pero esta opción ha sido colocada dentro de `SQLBulkOperations` [33].

Actualización (*update*) Llamar a `SQLSetPos` con un número de fila y el parámetro `SQL_UPDATE` provoca que la fila se actualice en el DBMS en base a los valores que hay escritos en las variables enlazadas a las columnas [33]. De esta manera, una aplicación enlaza unas variables a unas determinadas columnas del conjunto de resultados, escribe en estas variables los valores que la aplicación desee, y envía a `SQLSetPos` con `SQL_UPDATE` y el número de fila, que será la fila donde se quiere reemplazar los valores actuales asociados a esa fila y dichas columnas por el de las variables asociadas a dichas columnas, con efecto en la fuente de datos.

MyODBC lo que hace es construir una consulta `UPDATE` en sintaxis SQL en base a los valores de las variables, y la envía a MySQL Server como una consulta cualquiera.

Observamos que tenemos que reemplazar la llamada a MySQL por la llamada a DES. Pero también tenemos que tener cuidado con cómo se construye la consulta, puesto que no será completamente compatible con DES.

DES soporta la sintaxis SQL: con el comando `/sql`, DES cambia el intérprete por un intérprete SQL puro, de modo que todas las consultas se analizan y se ejecutan en SQL. Sin embargo, algunas de las consultas que se enviaban desde MyODBC introducían peculiaridades relativas a MySQL.

Nuestra decisión ha sido ceñirnos a la sintaxis que define el intérprete por defecto

[18]Un conector ODBC no tiene por qué implementarlo: un usuario debe llamar a `SQLGetInfo` bajo la opción `SQL_BOOKMARK_PERSISTENCE` para consultarlo. En [12] se tratan los marcadores en el capítulo 11. *Retrieving Results (**Advanced**)*; se considera una funcionalidad avanzada.

de DES para las operaciones *insert*/*update*/*delete*, pues ya se presentan en formato SQL (sin necesidad de acudir al modo DES-SQL) y por tanto son muy similares a las de MyODBC.

Así, construiríamos la consulta `UPDATE` en base a la siguiente sintaxis [7, sección 4.2.5.3]:

```
UPDATE TableName [[AS] Identifier] SET Att1=Expr1,...,AttN=ExprN
[WHERE Condition]
```

Tuvimos que hacer las modificaciones pertinentes al proceso de construcción de la consulta en MyODBC.

Recuperación (*refresh*) Si se llama a `SQLSetPos` con `SQL_REFRESH`, se actualizan las variables de aplicación asociadas a las columnas en base a los valores actuales del conjunto de resultados, en la fila especificada por el usuario [33]. Se trataría, pues, de una especie de «inversa del `UPDATE`», del caso anterior. No existe ninguna interacción con el DBMS y por tanto no hay nada que modificar de MyODBC.

Eliminación (*delete*) Cuando se especifica `SQL_DELETE` en `SQLSetPos`, se elimina la fila especificada por el usuario con efecto en la fuente de datos [33]. Las filas en el conjunto de resultados vienen numeradas, pero en DES no existe una indexación «nativa» de las filas[19]. Es por esto que tenemos que construir una consulta `DELETE` en base a todos los atributos que encontremos en la fila determinada. Construimos la consulta *delete* bajo la sintaxis siguiente [7, sección 4.2.5.2]:

```
DELETE FROM TableName [[AS] Identifier] [WHERE Condition]
```

Sin embargo, existe un problema con esta sintaxis: se eliminarán todas las filas duplicadas que respondan a los valores de la fila a eliminar. MyODBC prevee este escenario y le coloca como sufijo `LIMIT 1` antes de enviar la consulta. Ciertamente, la sintaxis de las versiones modernas de MySQL posibilitan colocar la palabra clave `LIMIT` [34]:

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name [[AS] tbl_alias]
    [PARTITION (partition_name [, partition_name] ...)]
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

Para sortear este problema ejecutamos el siguiente algoritmo:

[19] Las bases de datos relacionales en la mayoría de DBMS están diseñadas para trabajar en conjuntos de datos que no tienen un orden preestablecido.

1. Hallamos el número de filas duplicadas con una sentencia del tipo `select COUNT (*)from nombre_tabla WHERE ...` Almacenamos el resultado en una variable, con resultado digamos n.
2. Eliminamos todas las filas duplicadas con nuestro `DELETE`.
3. Insertamos $n-1$ veces la fila eliminada.

Caso especial general a las operaciones: fila cero Si se especifica como argumento el número de fila 0, se establece que la operación especificada se ejecute sobre todas las filas del conjunto de resultados [33]. MyODBC lo único que hace es iterar sobre el número de filas, aplicando de una en una la operación deseada. DES tampoco ofrece una forma más compacta de operar por filas, y por tanto también procedemos a hacer la operación fila por fila.

`SQLBulkOperations` Esta es una función que, como su nombre indica, que permite realizar operaciones «a granel». Admite como argumento una operación, que puede ser `SQL_ADD`, `SQL_UPDATE_BY_BOOKMARK`, `SQL_DELETE_BY_BOOKMARK` o `SQL_FETCH_BY-_BOOKMARK` [35].

Añadir (*add*) Si se indica `SQL_ADD`, se insertan nuevas filas a granel. Para ello, previamente se debe especificar el número de filas que se deben insertar modificando el atributo `SQL_ATTR_ROW_ARRAY_SIZE` del elemento sentencia, después enlazar los datos a insertar con `SQLBindCol` con un array de tamaño igual al número de filas especificado previamente, y finalmente llamando a `SQLBulkOperations` con `SQL_ADD` [35].

En cualquier caso, todo se traduce al final en construir una consulta de tipo `INSERT` por cada fila nueva a insertar. Como ocurría con los `DELETE` y los `UPDATE`, hemos adaptado ligeramente el código para construir una consulta de tipo `INSERT` exactamente igual a como lo espera DES [7, sección 4.2.5.1]:

```
INSERT INTO TableName[(Col1,...,ColN)] VALUES (Expr1,...,ExprN) [,
..., (Expr1,...,ExprN)]
```

Caso especial general a las operaciones: marcadores Hemos visto que el resto de opciones consisten en hacer *update*, *delete* o *fetch* por marcador. Esto se traduce en que, para cada fila a la que hayamos añadido marcador en el pasado, se actualice/elimine/recupere respectivamente. Nuevamente, esto se tiene que hacer fila por fila. Todo esto se implementa fácilmente sobre lo ya expuesto.

3.5.5. Funciones de catálogo (metadatos)

Las funciones de catálogo, o de metadatos, son una serie de funciones API que nos permiten comprobar ciertos aspectos sobre los esquemas y estructura de las bases de datos, o algunos aspectos del propio DBMS.

Consideramos que a este grupo forman parte `SQLColumns`, `SQLColumnPrivileges`, `SQLGetTypeInfo`, `SQLForeignKeys`, `SQLPrimaryKeys`, `SQLProcedures`, `SQLProcedureColumns`, `SQLSpecialColumns`, `SQLStatistics`, `SQLTables` y por último `SQLTablePrivileges`.

Vamos a recorrerlas en las siguientes secciones en base a la categoría a la que englobamos. Antes de nada, tenemos que explicar unos preliminares que serán comunes a todas estas funciones de catálogo.

3.5.5.1. Preliminares

Catálogos, esquemas y tablas En muchas de estas funciones se nos pide que especifiquemos, además de el nombre de una tabla, su catálogo y esquemas asociados.

Introducimos en este momento la cuestión de los catálogos y esquemas. Los catálogos, esquemas y relaciones conforman una jerarquía de tres niveles, en cuya cima se encuentran los catálogos, que contienen esquemas, y cada esquema constituye una colección de relaciones[20] [36, p. 163].

Estos conceptos en la práctica no son del todo estándar: cada DBMS los interpreta de determinadas maneras. MyODBC encuentra problemática la distinción catálogo / esquema: en MySQL tanto *catalog*, como *schema* como *database* son conceptos equivalentes en determinadas situaciones, y por tanto propensos a confundirse en determinados contextos. MyODBC opta por dejar al usuario que modifique las opciones `NO_CATALOG` y `NO_SCHEMA` en la fuente de datos con el fin de que sepa qué interpretaciones toma MyODBC en base a lo que se indica en la documentación (véase [37]).

Insistimos en que como estos conceptos no son estándar del todo, cada DBMS toma la decisión de interpretarlos de acuerdo a un estándar que ellos mismos proponen en la documentación.

Veamos qué ocurre en DES. Primero, los tipos de bases de datos a las que DES puede acceder son:

- **Base de datos DES**: son bases de datos del ámbito de DES. Es decir, o bien la base de datos por defecto al iniciar DES, o bien una guardada como archivo. Las guardadas como archivo se cargan en DES por medio del comando `/restore_ddb` o `/restore_ddb` *`Filename`*, y pasan a reemplazar la base de datos de DES vigente. Es decir, **ligada a una sesión DES, solo puede haber una base de datos cargada a la vez**.

  Tanto si es la base de datos por defecto como una cargada de archivo, al consultar el nombre de la base de datos actual se nos informa de que se llama `$des`.

- **Bases de datos externas**: son bases de datos a las que hemos accedido por medio de una conexión ODBC desde DES al conector ODBC de un DBMS particular. Es decir, nos hemos conectado a una base de datos por medio del comando `/open_db`; por ejemplo, `/open_db mysql`. En este caso, al consultar

[20]Dentro de las relaciones, en las funciones de catálogo solo trataremos las tablas.

el nombre de esta base de datos, se llamará como la fuente de datos a la que nos hemos conectado; en el caso de habernos conectado mediante `/open_db mysql`, la base de datos se llamará `mysql`.

Podemos tener cargado en la sesión tanto la base de datos de DES (siempre presente) como otras bases de datos externas. En la figura 3.9 puede observarse cómo se puede conocer en DES el listado de estas bases de datos con el comando `/show_dbs`.

```
DES> /show_dbs
$des
access
csv
db2
excel
mysql
oracle
postgresql
sqlserver
```

Figura 3.9: Ejemplo de salida de la llamada a `/show_dbs` [7, sección 5.1.3].

Nota 1 *Parece algo confuso hablar de bases de datos en este contexto. Estamos llamando «base de datos» a una conexión a un DBMS externo, lo cual no es preciso del todo: dentro una conexión a un DBMS externo pueden coexistir varias bases de datos. Mismamente, en MySQL, se permite dentro de una misma conexión (usuario-contraseña) crear y usar varias bases de datos que coexisten, véase el comando de MySQL `CREATE DATABASE`. Suponiendo que en MySQL hayamos creado dentro de un mismo usuario las bases de datos `bdd1` y `bdd2`, DES podría, dentro de esta conexión, llamar a `select from bdd1.tabla` así como `select from bdd2.otra_tabla`.*

De este modo, cuando hablamos de «bases de datos externas» en el contexto de DES, en realidad deberíamos decir «conexiones», aunque estas conexiones luego redunden en que efectivamente trabajemos con bases de datos externas. Hacemos notar este detalle para clarificar algunos de estos conceptos.

Otro aspecto a tener en cuenta es que DES no tiene una visión global sobre las conexiones. Solo puede encontrarse en el marco de una conexión determinada (por ejemplo, si hacemos referencias a tablas de otras conexiones en las que no estamos, obtengo un error). Con `/current_db` se puede averiguar en qué conexión nos encontramos actualmente (`$des`, `access`...) y podemos cambiar de conexión (i.e., cambiar el contexto) con `/use_db` *`nombre`*.

Con todos estos detalles que hemos recogido de DES, observamos que el concepto que encaja de una forma más natural con el del catálogo es el de conexión.

Con respecto a los esquemas, DES no proporciona soporte para esquemas más que el que se ofrece por defecto al ejecutar `/dbschema` (o en SQL `DESCRIBE TABLES`, también en DES) sobre las distintas conexiones. Por tanto, haremos como MyODBC aplicando restricciones al usuario: en nuestro caso, forzar de antemano a que no se pueda trabajar con esquemas.

En conclusión, dentro de las funciones de metadatos admitiremos el parámetro de catálogo, que alude a una conexión (sea `$des` o una base de datos externa), el nombre de tabla, e ignoraremos el argumento de esquema.

En algunas de estas funciones de metadatos especificar una conexión externa involucrará trabajar dentro del contexto de dicha conexión: por tanto, tendremos que utilizar el comando `/use_db` *conexión_externa* y tras finalizar las operaciones restauraremos el estado llamando a `/use_db` *conexión_anterior*. El proceso completo se realizará con el cerrojo consulta adquirido, con el fin de «atomizar» la operación. En adelante omitiremos repetir este detalle de la implementación.

Patrones de búsqueda En algunas de estas funciones de catálogo se pueden introducir patrones de búsqueda en lugar de los nombres completos de tabla, esquema o catálogo, posibilidad introducida en ODBC 3.x. Las reglas de los patrones de búsqueda se pueden consultar en el manual (véase [38]).

Atendiendo a que únicamente permitimos introducir nombre de tabla, podríamos introducir como nombre `departamento%`. En este escenario, la función a la que se llamara consideraría todas las tablas cuyo nombre tenga como prefijo `departamento`. Para obtener la lista de nombres de tabla sobre los que buscar, se hace una llamada previa a `/dbschema`. Por cada tabla coincidente haría lo propio de esa función a la que habríamos llamado. El conjunto de resultados final consiste en juntar todas las filas de los conjuntos de resultado obtenidos en cada una de estas combinaciones.

Todo este proceso, naturalmente, con el cerrojo de consulta adquirido para «atomizar» esta operación.

En adelante solo hablaremos de introducir nombres de tabla, sin referirnos a esta posibilidad de búsqueda (en algunas funciones se permite y en otras no, pero señalar dónde es irrelevante para este documento).

3.5.5.2. Columnas

`SQLColumns` Se devuelve la lista de columnas dados patrones de búsqueda de un nombre de tabla y/o un nombre de columna [39]. Para ello, dentro de la conexión especificada se llama a `select * from` *nombre_tabla* para acceder a los nombres de las columnas de la tabla. Si se especifican nombres de columna en los parámetros, se filtran del conjunto del resultado las columnas no coincidentes con las especificadas.

Internamente, DESODBC extrae la información de la salida de esta consulta *select* y construye el conjunto de resultados esperado según la especificación de `SQLColumns`. Es necesario llamar a `select * from` *nombre_tabla* en lugar de un simple `/dbschema` *nombre_conexión*`:`*nombre_tabla* puesto que en el conjunto de

resultados se devuelve información relativa a los datos como, por ejemplo, el número máximo de decimales significativos que aparecen en datos de la columna.

`SQLColumnPrivileges` Esta función recibe como parámetro el nombre de una tabla y devuelve información acerca de los privilegios de cada una de sus columnas [40]. No existe el concepto de privilegios en DES, y por tanto marcamos la función frente al usuario como «no implementada».

Nota 2 *Puede preguntarse qué ocurre si en esta función se piden los privilegios dentro de un catálogo que aluda a una conexión a un DBMS que sí soporte privilegios. Lo que ocurre en este caso es que a ojos de DES, no existen: no existen comandos DES para acceder dentro de una conexión así a este tipo de características que otros DBMS implementan, como privilegios, procedimientos, índices... DES permite conocer de otras conexiones lo que podría conocer de una base de datos* `$des`[21]*. Por tanto, el cliente no debería interpretar una conexión desde DES a otro DBMS como una conexión «real», sino como que DES accede a los datos en sí de esta base de datos, y los interpreta y estructura bajo la manera de proceder de DES.*

Por tanto, cuando se haya especificado una conexión externa y la función ODBC permita potencialmente devolver este tipo de elementos externos como parte del conjunto de resultados, se señalará por medio de un mensaje `SQL_SUCCESS_WITH_INFO` *que se ha podido omitir información relativa a índices, procedimientos, número de páginas o privilegios. En las funciones no implementadas como* `SQLColumnPrivileges`*, si se especifica una conexión externa se adjunta en el mensaje de error lo propio.*

Una vez explicado este problema, omitiremos explicarlo de nuevo cuando lleguemos a otras funcionalidades que DES tampoco implementa, como procedimientos e índices.

`SQLSpecialColumns` `SQLSpecialColumns` permite saber para una tabla cuál es el conjunto de columnas «especiales», entendiendo por especiales las que identifican de forma única las filas de la tabla (claves primarias), y las que se actualizan automáticamente cuando se efectúa una transacción que haya modificado algún valor de estas columnas [41]. DES no soporta transacciones y por tanto devolvemos únicamente las claves primarias. Este proceso será prácticamente idéntico al que se haga en `SQLPrimaryKeys`, y por tanto comentaremos cómo se obtienen las claves primarias en dicho apartado.

3.5.5.3. Claves

Las funciones `SQLPrimaryKeys` y `SQLForeignKeys` permiten qué columnas de una tabla están asociadas a claves primarias y foráneas respectivamente.

[21]Podemos enviar a las conexiones externas consultas propias de dichos DBMS para que lean y manipulen este tipo de elementos. No obstante, a pesar de poder recurrir a esta posibilidad, nos estamos refiriendo a que DES no presenta comandos ni características propias para acceder «de forma nativa» a estas características que no están presentes en bases de datos DES. Como cada DBMS tendrá su conjunto particular de consultas para acceder a estos elementos, optamos por restringir por diseño la consulta de los mismos.

El comando `/dbschema` *`nombre`* permite conocer dado un nombre de tabla, entre otras cosas, sus claves primarias y foráneas. En la figura 3.10 se observa un ejemplo de salida de este comando. Analizando esta salida es como podremos implementar ambas funciones.

```
DES> /dbschema registration

Info: Database '$des'
Info: Table:
 * registration(student:varchar(45),course:varchar(11),pass:int)
    - NN: [student,course]
    - PK: [student,course]
    - FK: registration.[course] -> courses.[id]
    - FK: registration.[pass] -> prueba.[id]
```

Figura 3.10: Vista de las claves primarias y foráneas de una tabla en DES.

Apuntamos que no se pueden conocer las claves primarias y foráneas de tablas de conexiones externas (no se subrayan dentro del `dbschema`, probablemente por la imposibilidad de conocerlas con un procedimiento que funcione para todos los DBMS). Cuando el usuario especifique en el campo de un catálogo una conexión distinta de `$des`, notificaremos esta limitación.

Con `SQLPrimaryKeys` se espera únicamente que, dado un nombre de tabla, se devuelvan los nombres de columna de las claves primarias, formateados adecuadamente [42]. La implementación de `SQLForeignKeys` es más compleja pues admite las siguientes posibilidades como argumento [43]:

1. Introducir una tabla en un campo `PKTableName`.

   Se devuelven las claves primarias de dicha tabla y, por cada clave primaria, las claves foráneas que aluden a ella.

2. Introducir una tabla en un campo `FKTableName`.

   Se devuelven las claves foráneas de dicha tabla que apuntan a claves primarias de otras tablas.

3. Introducir una tabla en ambos campos.

   Fusión de ambos casos: se devuelven las claves foráneas de la tabla de nombre `FKTableName` que apuntan a claves primarias de la tabla de nombre `PKTableName`.

Subrayamos estos detalles puesto que requieren una implementación más delicada. En particular, tendremos que conocer las claves primarias y foráneas de todas las tablas; esto se puede conseguir llamando a `/dbschema`, pues si no se especifica tabla, se muestra el esquema de todas las relaciones. Por tanto, primero extraemos

la información de la salida en formato TAPI de `/dbschema` y la introducimos en las estructuras de datos adecuadas del conector (véase el listado 3.19), y después consultaremos información de la misma de acuerdo a las necesidades especificadas en los argumentos[22].

```
// driver/driver.h (DESODBC)
struct ForeignKeyInfo {
  std::string key;
  std::string foreign_table;
  std::string foreign_key;
};

struct DBSchemaTableInfo {
  // Unordered maps name -> column index
  std::unordered_map<std::string, int> columns_index_map;
  std::unordered_map<std::string, Type> columns_type_map;
  std::vector<std::string> primary_keys;
  std::vector<ForeignKeyInfo> foreign_keys;
  std::vector<std::string> not_nulls;

  std::string name;
};
```

Listado 3.19: Estructuras que almacenan información del `dbschema`.

3.5.5.4. Procedimientos

El estándar especifica que `SQLProcedures` «*devuelve la lista de procedimientos (procedures) que estén almacenados en una fuente de datos*»[23] [44]. Se menciona en esta referencia que los *procedures* son, en abstracto, procedimientos que se ejecutan (por ejemplo funciones que admiten unos argumentos y devuelven una salida). Por norma general, son sentencias SQL. La utilidad de las mismas es poder aludir a un procedimiento completo por medio de un *alias*, y así no repetir código. En la figura 3.11 puede verse un ejemplo de un procedimiento en MySQL.

Los procedimientos tal y como se entienden en el estándar SQL no se encuentran presentes en DES (tampoco en el modo `/sql`). Existe cierta semejanza semántica entre el concepto de *procedure* como objeto ejecutable y las reglas Datalog que pueden establecerse en una base de datos DES. Técnicamente, podríamos implementar `SQLProcedures` en base a esta convención. Optamos por no hacerlo por los siguientes motivos:

[22]Atomizamos esta operación adquiriendo el cerrojo de consulta durante todo el proceso, de modo que los datos conservan la consistencia. Procedemos igual en otras situaciones donde hay un riesgo de pérdida de la consistencia, pero obviaremos señalarlo.

[23]Traducciones propias.

```
mysql> CREATE PROCEDURE citycount (IN country CHAR(3), OUT cities
                                   INT)
       BEGIN
         SELECT COUNT(*) INTO cities FROM world.city
         WHERE CountryCode = country;
       END
Query OK, 0 rows affected (0.01 sec)

mysql> CALL citycount('JPN', @cities); -- cities in Japan
Query OK, 1 row affected (0.00 sec)
```

Figura 3.11: Ejemplo de procedures en MySQL. Extraído de [45].

1. A fin de cuentas, `SQLProcedures` está definido dentro del contexto estricto de SQL, como indica su propio prefijo. El usuario espera obtener «procedimientos» dentro de cómo están definidos en el estándar SQL, que se sale por completo de la sintaxis y de las características de las reglas Datalog.
2. Las reglas Datalog definen predicados de un lenguaje de propósito específico, no general; por tanto, no se llaman «*procedures*» ni procedimientos bajo ningún estándar.

DES asimismo presenta un lenguaje de *scripting* Turing-completo mediante el cual podemos aproximarnos al concepto de procedimientos. Sin embargo, también la descartamos por el mismo motivo de falta de adscripción estricta al estándar SQL en cuanto a lo que se entiende por *procedures*.

Optamos por notificar esta función como «no implementada», junto con `SQLProcedureColumns`, una función que también trabaja sobre los procedimientos.

3.5.5.5. Tablas

Nos referimos a las funciones `SQLTablePrivileges` y `SQLTables`. La primera función la marcamos como no implementada; ya establecimos que no existe el concepto de privilegios en DES. `SQLTables` simplemente devuelve la lista de tablas según patrones de búsqueda colocados en el argumento de catálogo, esquema, tabla y tipo de relación (tabla o vista) [46].

3.5.5.6. Tipos

El estándar ODBC distingue dos tipos de tipos de datos: tipos de datos SQL y tipos de datos C.

Los tipos de datos SQL son los tipos de datos asociados a los tipos de datos del DBMS que soporta SQL. Esto es, el estándar SQL define una serie de tipos de datos que después cada DBMS se encarga de imitar de entre todos los que SQL ofrece.

Por otro lado, los tipos de datos C se usan en la aplicación con el fin de especificar tipos de datos a variables de tipo aplicación. Son simplemente tipos de datos utilizados en el código C de la aplicación, escrita en C/C++. Por tanto, estos tipos de datos vienen definidos por la especificación de C.

Esto viene en la línea de los *buffer* en cuanto estándar acordado entre aplicación y conector; los tipos de datos C son tipos acordados entre ambos extremos (esto es posible puesto que tanto la aplicación como el conector conocen C/C++).

El «acuerdo» proviene de que ambos conocen el estándar ODBC, que establece una correspondencia entre el alias de «tipo de dato C» y el tipo de dato C real. Véanse los ejemplos de la tabla 3.2.

C type identifier	ODBC C typedef	C type
`SQL_C_CHAR`	`SQLCHAR *`	`unsigned char *`
`SQL_C_FLOAT`	`SQLREAL`	`float`
`SQL_C_BIT`	`SQLCHAR`	`unsigned char`

(*ODBC C typedef* corresponde a un alias del tipo C —estilo `using`—, para no tratar con este directamente).

Tabla 3.2: Algunos ejemplos de correspondencia entre tipos de datos C y «tipos de datos C ODBC», extraídos de [47].

Por ejemplo, si aplicación o conector se encuentran con que un *buffer* tiene asignado un tipo de datos de identificador numérico `SQL_C_CHAR` (están definidos bajo unos números concretos en los archivos de encabezado de ODBC y SQL), saben que tienen que tratar la información de dicho buffer como `SQLCHAR *`, alias de `unsigned char *`.

El estándar define una correspondencia entre tipos de datos SQL y los tipos de datos C, que el conector debe encargarse de implementar. Supongamos el siguiente ejemplo completo:

Una aplicación ha ejecutado una consulta. Se encuentra con el cursor posicionado en una fila determinada. Quiere conocer el valor que existe dentro de la fila en una columna determinada. Llama, por tanto, a la función `SQLGetData`, cuya cabecera puede observarse en el listado 3.20.

```
1 SQLRETURN SQLGetData(
2      SQLHSTMT StatementHandle,
3      SQLUSMALLINT Col_or_Param_Num,
4      SQLSMALLINT TargetType,
5      SQLPOINTER TargetValuePtr,
6      SQLLEN BufferLength,
7      SQLLEN * StrLen_or_IndPtr);
```

Listado 3.20: Cabecera de la función ODBC API `SQLGetData` [48].

En TargetType, la aplicación especifica qué tipo de datos C espera encontrar.

Supongamos que quiere obtener el valor en una columna de tipo fecha. Haciendo una serie de llamadas previas, averiguó que el tipo interno de esta columna es SQL_TYPE_DATE.

El estándar ODBC establece que un SQL_TYPE_DATE puede convertirse tanto en SQL_C_CHAR como en SQL_C_TIMESTAMP [49], cuya estructura de datos correspondiente puede observarse en el listado 3.21.

```
struct tagTIMESTAMP_STRUCT {
   SQLSMALLINT year;
   SQLUSMALLINT month;
   SQLUSMALLINT day;
   SQLUSMALLINT hour;
   SQLUSMALLINT minute;
   SQLUSMALLINT second;
   SQLUINTEGER fraction;
} TIMESTAMP_STRUCT;
```

Listado 3.21: Estructura asociada a SQL_C_TIMESTAMP. Extraído de [47].

Ejemplos de transformaciones de tipo SQL a tipo C pueden observarse en la tabla 3.3.

SQL type id.	***SQL data value***	***C type id.***	***Buf. length***	***TargetValuePtr**
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0
SQL_TYPE_DATE	1992-12-31	SQL_C_TIMESTAMP	ignored	1992,12,31, 0,0,0,0

Tabla 3.3: Algunos ejemplos de transformaciones concretas entre tipos SQL-C extraídos de [49].

La aplicación finalmente desea tratar esta fecha como cadena, de forma que introduce en TargetType el valor SQL_C_CHAR, de forma que según la tabla 3.2 deseará obtener un valor de tipo SQLCHAR*, alias de unsigned char*; y en TargetTypeValuePtr coloca el puntero a un *buffer* de tipo SQLCHAR*.

Al llamar a la función, el conector procede internamente a hacer un *casting* de un valor de tipo fecha del DBMS a un valor de tipo cadena, y finalmente escribe este resultado en el puntero del *buffer* pasado como argumento. En el listado 3.22 puede observarse el ejemplo concreto.

```
  ok_sql(hstmt, "create or replace table test(d date)");
  ok_sql(hstmt, "insert into test values(cast(\'2025-02-13\' as date))");
  ok_sql(hstmt, "select * from test");

  ok_stmt(hstmt, SQLFetch(hstmt));
```

```
6
7   SQLCHAR datebuf[TEST_BUFFER_SIZE];
8   SQLLEN len;
9
10  ok_stmt(hstmt, SQLGetData(hstmt, 1, SQL_C_CHAR, datebuf, sizeof(datebuf),
11                 &len));
12
13  is_str("2025-02-13", datebuf, len);
14
15  return OK;
```

Listado 3.22: Aplicación de tipo test (véase el apartado 3.6) que comprueba el *casting* correcto al tipo real de C por medio de llamadas a DESODBC. Primero inserta un valor de tipo fecha en DES y después la recupera en tipo cadena.

Hablemos de qué ocurre en nuestro conector. Recordemos que las transformaciones de tipos C a SQL y viceversa deben realizarse desde el conector: por suerte, podemos reutilizar todo el código de MyODBC a este respecto. Sin embargo, hemos tenido que rehacer el aspecto de MyODBC relacionado con los tipos MySQL, pues difieren de los de DES.

Empecemos considerando qué correspondencia establecer entre los tipos de datos definidos en DES y los tipos SQL. DES considera los tipos de datos válidos en tuplas que pueden observarse en la tabla 3.4.

Type	***Meaning***
`varchar, string`	*String of unbounded length*
`char(N),` *`varchar(N)`*	*String with length up to N*
`char`	*String with length 1*
`integer, int`	*Integer number*
`float, real`	*Real number*
`date`	*Date expressed as* *`date(Year,Month,Day)`*
`time`	*Time expressed as* *`time(Hour,Minute,Second)`*
`datetime,` *`timestamp`*	*Timestamp expressed as* *`datetime(Year,Month,Day,Hour,Minute,Second)`*

«Precision and range depend on the underlying Prolog system.»

Tabla 3.4: Tipos de datos DES [7, sección 4.1.16.1][24].

[24]El manual también lista en [7, sección 4.2.4.1] los tipos de datos que se pueden especificar en tiempo de creación de tablas, e incluyen otros adicionales a los de tabla: los `NUMERIC` y `DECIMAL`, que además admiten parámetros. No obstante, se traducen internamente a los tipos presentados en la tabla: estos son los tipos de datos que constituyen las restricciones de integridad. El conector solo lidiará con este último conjunto de tipos de datos, pues las consultas sobre los tipos de datos desde el conector solo se dan sobre tablas ya creadas (y por tanto, sus tipos ya traducidos a los de la tabla).

En base a esta última cita textual del manual, tenemos que ver qué precisiones y rangos se establecen en los dos sistemas Prolog posibles sobre los que existen sendas versiones para DES: `SICStus Prolog` o `SWI-Prolog`.

`SICStus Prolog` considera [50, sección 4.7.1]:

1. El rango de los enteros es ilimitado.

2. Los tipos de punto flotante ocupan 64 bits y sus rangos se corresponden al de `double` de C, típicamente `[4.9e-324, 1.8e+308]`.

`SWI-Prolog` suscribe las mismas consideraciones. Sin embargo, el rango de los enteros es ilimitado solo cuando `SWI-Prolog` se compila con *GNU multiple precision arithmetic library (GMP)*; de no serlo, los enteros son de 64 bits [51, sección 4.27]. Yendo al caso prototípico en el que se usa la versión ya compilada de DES con `SWI-Prolog` confirmamos que el rango de enteros es ilimitado.

En cualquier caso, es un problema el hecho de considerar enteros de rango ilimitado: tal cosa no se encuentra de manera nativa en C, y por tanto, no existe un tipo de datos C asociado. En la última versión de ODBC, ODBC 3.8, se permiten definir tipos C específicos en el conector [52]: sin embargo, en este proceso el rango del tipo debe estar acotado entre una serie de valores.

Una implementación de enteros ilimitados que sortee este problema podría ser materia para trabajos futuros. Por tanto, por ahora consideraremos que los enteros son de 64 bits. De esta manera, además seremos conservadores con el caso residual en el que el cliente use un DES compilado con una versión de `SWI-Prolog` compilada sin la biblioteca GMP.

La correspondencia final de tipos DES a tipos SQL se presenta en la tabla 3.5.

3.5.5.7. Estadísticas

Nos referimos a la función `SQLStatistics`, que espera que dado un nombre de tabla se devuelvan una serie de características de la tabla referida [54]. La mayoría de ellas se refieren a índices, que no están soportados por DES; sin embargo, se puede devolver solo información de la tabla en sí especificando que se devuelve un conjunto de resultados de tipo `SQL_TABLE_STAT`. En este caso, se devolverá:

1. `CARDINALITY`: en nuestro caso, el número de filas de la tabla.

2. `PAGES`: en nuestro caso, el número de páginas empleadas para almacenar la tabla.

Este último valor es opcional y de hecho no lo devolveremos, puesto que no podemos conocer este valor por medio de simples peticiones a DES.

Tipo de datos DES	**Id. de tipo SQL**	**Tipo de datos SQL**	**Precisión (bits)**	**Rango**
`varchar` / `string`	`SQL_LONGVARCHAR`	`LONG VARCHAR`	—	—
`char(`N`)`	`SQL_CHAR`	`CHAR(`N`)`	—	—
`varchar(`N`)`	`SQL_VARCHAR`	`VARCHAR(`N`)`	—	—
`char`	`SQL_CHAR` (*)	`CHAR(1)`	—	—
`integer` / `int`	`SQL_BIGINT`	`BIGINT`	19 (***)	$[-2^{63}, 2^{63}-1]$
`float` / `real`	`SQL_DOUBLE` (**)	`DOUBLE PRECISION`	53	$[10^{-308}, 10^{308}]$ (en abs.)
`date`	`SQL_TYPE_DATE`	`DATE`	—	—
`time`	`SQL_TYPE_TIME`	`TIME(`p`)` (****)	—	—
`datetime` / `timestamp`	`SQL_TYPE_STAMP`	`TIMESTAMP(`p`)`	—	—

(*) En SQL no existe el concepto de `char` tal y como se pueda comprender en C o DES, como «único caracter». Se implementará entonces bajo la perspectiva de «cadena de longitud 1».
(**) En DES `float` es sinónimo de `real`, cosa que no ocurre en SQL. Los `float` y `real` de DES aluden al tipo de datos DOUBLE de SQL.
(***) Sería de precisión 20 si admitiéramos un `double` de tipo `unsigned`, pero no tiene cabida aquí. En DES no hay tipos sin signo.
(****) En TIME y TIMESTAMP se considera como parámetro dependiente p, que indica la precisión de los segundos. El propio estándar de SQL considera que este parámetro es opcional, y no tiene sentido en DES, por lo que lo ignoraremos.

Tabla 3.5: Correspondencia de tipos de datos DES con tipos de datos SQL. Las precisiones y los rangos pueden consultarse en [53].

3.5.6. Sistema de gestión de errores

3.5.6.1. Errores en el conector

Un *software* de calidad debe implementar una gestión de errores suficientemente exhaustiva. Por fortuna, ODBC nos ayuda en esta tarea. Las funciones ODBC API devuelven un valor de tipo `SQLRETURN`. Este tipo comprende los valores autoexplicativos `SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, `SQL_ERROR`, etcétera.

Cuando el usuario ejecuta una función en la que obtuvo valores que conlleven una información, como por ejemplo `SQL_SUCCESS_WITH_INFO` o `SQL_ERROR`, si quiere conocer el mensaje asociado debe llamar a la función ODBC API `SQLGetDiagRec`. Esta función devuelve en un puntero la cadena asociada al mensaje del campo de diagnóstico del manejador especificado [55].

Por tanto, en las implementaciones de las funciones, siempre que nos encontremos

en una situación susceptible de ser informada al usuario, tendremos que rellenar el campo de diagnóstico del manejador con el que estemos trabajando (de conexión, sentencia...). Esto nos permite informar al usuario de los errores cuando suceden.

Veamos un ejemplo de gestión de errores. El código del listado 3.23 trata la creación de la tubería de salida en Windows, dentro de una función `DBC::create_pipes`.

```
// driver/connect.cc (DESODBC)
SQLRETURN DBC::create_pipes() {
    ...
    if (!CreatePipe(&this->driver_to_des_out_rpipe,
                  &this->driver_to_des_out_wpipe, &des_pipe_sec_attr,
                  des_pipe_buf_size)) {
    CloseHandle(this->driver_to_des_out_rpipe);
    CloseHandle(this->driver_to_des_out_wpipe);
    return this->set_win_error("Failed to create DES output pipe", true);
  }
  ...
```

Listado 3.23: Gestión de error al crear la tubería de salida en Windows, dentro de una función `DBC::create_pipes`.

Al llamar a `CreatePipe` (`namedpipeapi.h`), si obtenemos `false` implica error de esta función.

Por tanto, lo gestionamos primero cerrando por seguridad los manejadores de tubería, y posteriormente llamando a una función del `DBC` llamada `set_win_error`. Esta función está pensada para errores relativos a Windows específicamente: su tarea es escribir en el campo de diagnóstico del `DBC` el mensaje que hemos especificado en uno de sus argumentos, e incluir además en dicho mensaje la información de diagnóstico propia de Windows (con `FormatMessage` —`winbase.h`— sobre `GetLastError` —`errhandlingapi.h`—). Existe una función análoga para sistemas basados en Unix llamada `set_unix_error`; y otra serie de funciones de error genéricas (no relativos a errores de sistema operativo, sino a todos los demás) `set_error`.

En cualquier caso, la función de tipo `set_error` devuelve además el valor de `SQLRETURN` correspondiente: en este caso particular, `SQL_ERROR`. Por tanto, el flujo de `DBC::createPipes` se detiene en este punto, y el error es recogido por la función que le ha llamado. El error en este caso lo recibe una función `DBC::connect`, que inmediatamente detiene su flujo y devuelve el error a la siguiente función llamadora en la pila, la función ODBC API `SQLConnect`. Esta función recoge el error e inmediatamente lo devuelve, de forma que globalmente, el usuario ha obtenido un `SQL_ERROR` al llamar a `SQLConnect`. Si desea conocer el mensaje de diagnóstico, insistimos, debe llamar posteriormente a `SQLGetDiagRec`.

3.5.6.2. Errores en DES

Un conector ODBC también tiene como tarea capturar los errores producidos en el DBMS destino y gestionarlos apropiadamente. De esta forma, si en una función

API ODBC, digamos `SQLExecDirect`, ejecutamos una consulta referente a una tabla inexistente, nuestro DBMS lo notificará, y nuestra tarea es capturar dicho error y lanzar el pertinente `SQL_SUCCESS_WITH_INFO` y `SQL_ERROR`.

Por fortuna, DES presenta un formato de error general para los comandos en la TAPI, del que nos podemos servir para observar el diagnóstico del propio DES tras la correspondiente consulta (véase la figura 3.12).

```
$error
code
text
...
text
$eot
```

Figura 3.12: Mensaje genérico de error en TAPI [7, sección 5.18.1.2].

El campo `code` puede valer 0 (denotando un error de excepción), 1 (*warning*) o 2 (mensaje informativo) [7, sección 5.18.1.2]. Vemos que podemos hacer una correspondencia entre 0 y `SQL_ERROR`, y 1/2 con `SQL_SUCCESS_WITH_INFO`.

3.5.7. Versiones ANSI y Unicode del conector

MyODBC ofrece su conector ODBC en dos versiones: ANSI y Unicode.

MySQL entiende por Unicode una codificación que soporta *wide characters* (UTF-16). Sin embargo, usar el término «Unicode» como lo hace MyODBC no es del todo correcto, pues la versión de ANSI también admite Unicode, solo que UTF-8.

Tanto UTF-8 como UTF-16 pueden codificar cualquier caracter Unicode. Lo que ocurre es que UTF-8 utiliza «bloques» de 8 bits para ello, y UTF-16 de 16 bits. Por ejemplo, veamos cómo se representa un determinado caracter chino U+24B62 en las siguientes codificaciones en la tabla 3.6.

Codificación UTF-8	`0xF0 0xA4 0xAD 0xA2`
Codificación UTF-16	`0xD852 0xDF62`
Codificación UTF-32	`0x00024B62`

Tabla 3.6: Distintas codificaciones Unicode del caracter chino en cuestión. Extraído de [56].

Es por esto que pese a que puedan representar el mismo número de caracteres, UTF-16 es incompatible con UTF-8 y por eso MyODBC distingue entre estas dos versiones.

Haciendo las pruebas pertinentes, DES no soporta UTF-16 (sí UTF-8). Asimismo, solo procesa el alfabeto de las lenguas de Europa occidental: ñ, ö, á, à. Concluimos que DES soporta el conjunto de caracteres conocido como Latin1 (ISO/IEC 8859-1). Por suerte, MyODBC permite seleccionar charsets de entre una gran lista almacenados en una carpeta de MySQL entre los que se encuentra `Latin1`.

Nos encontramos en la tesitura de eliminar la versión Unicode o no. Realmente, las funciones de la API ODBC con sufijo `W` (aludiendo a las que usan caracteres de doble espacio, o en la terminología de MyODBC, «Unicode»), en la implementación de MyODBC no se usa jamás los caracteres de doble cadena `wchar_t`; el conector siempre se limita a transformarlos previamente a caracteres de cadena simple `char`[25]. Veamos el ejemplo presentado en el código del listado 3.24.

```
// driver/unicode.cc (DESODBC)
SQLRETURN SQL_API
SQLPrepareWImpl(SQLHSTMT hstmt, SQLWCHAR *str, SQLINTEGER str_len,
                bool force_prepare)
{
  STMT *stmt= (STMT *)hstmt;
  uint errors;
  SQLCHAR *conv= sqlwchar_as_sqlchar(stmt->dbc->cxn_charset_info,
                                     str, &str_len, &errors);
  /* Character conversion problems are not tolerated. */
  if (errors)
  {
    x_free(conv);
    return stmt->set_error("22018", NULL);
  }

  SQLRETURN rc = DESPrepare(hstmt, conv, str_len, false, force_prepare);
  x_free(conv);
  return rc;
}
```

Listado 3.24: Ejemplo de traducción de caracteres de doble espacio a cadenas de tipo `char`.

Por tanto, viendo la función `sqlwchar_as_sqlchar`, simplemente tenemos que cerciornarnos de que `stmt->dbc->cxn_charset_info` alude al charset Latin1.

La razón de ser de una versión «Unicode» para DES se limitará entonces a brindar la posibilidad que el usuario tenga la posibilidad de usar cadenas con carácter de doble espacio, que aunque pareciera innecesario pues no puede explotar el repertorio

[25]Esto no supone ninguna pérdida de información. Si bien `char` está pensado para almacenar caracteres de tipo ASCII (8 bits, en lugar de 16), internamente no ha habido pérdida de información; la cadena simplemente ocupará el doble. El único detalle es que si lo leemos como cadena de `char`, representará una cadena sin sentido. Tendremos que reinterpretar la información interna, que no se ha perdido, como cadena de `wchar`, cuando queramos interpretarla.

completo Unicode en DES, quizá le sea lo más cómodo por ciertas circunstancias de codificación dentro de su aplicación. Por fortuna, no tendremos que modificar nada del código.

3.6. *Testing*

MyODBC provee una cantidad muy numerosa de tests en su código fuente. Los tests consisten en 35 ficheros fuente, cada uno de ellos orientado a un aspecto determinado del conector, múltiples tests unitarios, del orden de mil líneas de código por fichero, y diseñado específicamente para MySQL. Es por esto que hemos podido reutilizar muy pocos. Procedemos a eliminar casi todos los tests y reutilizamos el esquema que MyODBC se emplea para lanzar tests, definido en un archivo de encabezado `odbctap.h`.

Cuando un fichero fuente incluye `odbctap.h`, automáticamente se produce una conexión a la fuente de datos, y por tanto los tests deben incluir solamente las operaciones específicas a lo que se pretende comprobar, facilitándonos el proceso de *testing*. En el listado 3.25 puede verse un ejemplo de test.

De esta forma, hemos generado en total 20 tests que nos han servido de utilidad para hacer depuración y asegurarnos de que los potenciales *bugs* más importantes no tomen lugar. La lista final de tests es la siguiente y comprueba:

- Envío de consulta `SELECT` simple y recuperación correcta de todo el conjunto de resultados con las funciones ODBC API pertinentes.
    - Para un cursor estándar (*scrollable static cursor*).
    - Para un cursor bloque (*block cursor*).
- Enlazamiento de parámetros.
- Uso de las variables de aplicación.
- Transformación de tipos.
- Ejecución de `SQLColumns`.
- Ejecución de `SQLGetTypeInfo`.
- Ejecución de funciones de metadatos sobre claves primarias (`SQLPrimaryKeys`) y foráneas (`SQLForeignKeys`).
- Ejecución de `SQLSpecialColumns`.
- Ejecución de `SQLStatistics`.
- Ejecución de `SQLTables`.
- Ejecución de operaciones `INSERT`, `REFRESH`, `DELETE`, `UPDATE` (`SQLSetPos`).

  - Para un cursor estándar (*scrollable static cursor*).
  - Para un cursor bloque (*block cursor*).

- Ejecución de operaciones sobre marcadores.
- Ejecución de operaciones «a granel» (*bulk operations*).
- Desconexión con `SQLDisconnect`.
- Liberación correcta de los recursos con `SQLFreeHandle`.
- Comprobación de gestión de errores.
- Obtención de atributos del conector y de elementos sentencia y conexión.

```
#include "odbctap.h"

DECLARE_TEST(basics)
{
  SQLLEN nRowCount;

  ok_sql(hstmt, "DROP TABLE IF EXISTS t_basic");
  ok_sql(hstmt,
         "CREATE TABLE t_basic (id INT PRIMARY KEY, name VARCHAR(20))");
  /* Insert 3 rows of data */
  ok_sql(hstmt, "INSERT INTO t_basic VALUES (1,'foo'),(2,'bar'),(3,'baz')")
      ;
  /* SELECT query */
  ok_sql(hstmt, "SELECT * FROM t_basic");
  /* Check if the number of cols is 3 */
  ok_stmt(hstmt, SQLNumResultCols(hstmt, &nRowCount));
  is_num(nRowCount, 3);
  ...
  return OK;
}

BEGIN_TESTS
  ADD_TEST(basics)
  ...
  END_TESTS

RUN_TESTS
```

Listado 3.25: Estructura de un fichero de test. Reutilizado de MyODBC y modificado para la exposición.

Cada uno de estos tests comprueba tanto que se obtienen las postcondiciones que se esperan como la ausencia de fugas de memoria. Se han diseñado de forma que comprobamos el funcionamiento correcto de todas las funciones de biblioteca ODBC API implementadas (aunque no demasiado exhaustivamente, dado el número inabarcable de casuísticas que pueden darse usando el conector).

3.7. Documentación

Una correcta documentación es un aspecto esencial de un *software* de calidad. La documentación de MyODBC consiste en unos pocos recursos en línea referidos principalmente al proceso de instalación[26] y comentarios en las cabeceras de las funciones implementadas.

En nuestro caso, este documento en sí constituye una forma de documentación, al tiempo que comentamos las funciones más importantes del conector: mantenemos los de MyODBC que mantengan su validez (y referenciamos su autoría, respetando sus derechos siguiendo las normas de GPLv2), y redactamos nuevos para las nuevas funciones y símbolos implementados más importantes. En el listado 3.26 puede verse un ejemplo de documentación de una función.

```
/* DESODBC:
    Removed the unnecessary fields of MyODBC's setup library
    regarding DESODBC's needs and introduced the new ones.
    Original author: MyODBC
    Modified by: DESODBC Developer
*/
// OnWMNotify - provides the tooltip control with the appropriate text
...
VOID OnWMNotify(WPARAM wParam, LPARAM lParam)
{
    LPTOOLTIPTEXT lpttt;
    int idCtrl;
```

Listado 3.26: Ejemplo de documentación de una función.

3.8. Sobre el desarrollo y depuración: software empleado

Dado que la estrategia de desarrollo consistió en hacer la implementación de Windows de forma completa, y posteriormente exportarlo a sistemas Unix modificando algunas pocos aspectos como el IPC, la mayoría del trabajo de desarrollo y depuración fue realizado en Windows. Pasamos a describir el software utilizado en ambos sistemas.

3.8.1. Independientes del sistema

Git Hicimos uso de Git, en Windows desde la utilidad GitHub Desktop. Con ello pudimos mantener un control de versiones limpio de toda la suite de DESODBC en un repositorio de GitHub.

[26] Nos referimos mayormente a [57].

GIMP GIMP es un software de edición de imágenes, de código abierto y multiplataforma. Esta aplicación nos sirvió para rediseñar el *banner* relativo a MySQL en la biblioteca de configuración adaptándolo para DES (véase la figura 3.13).

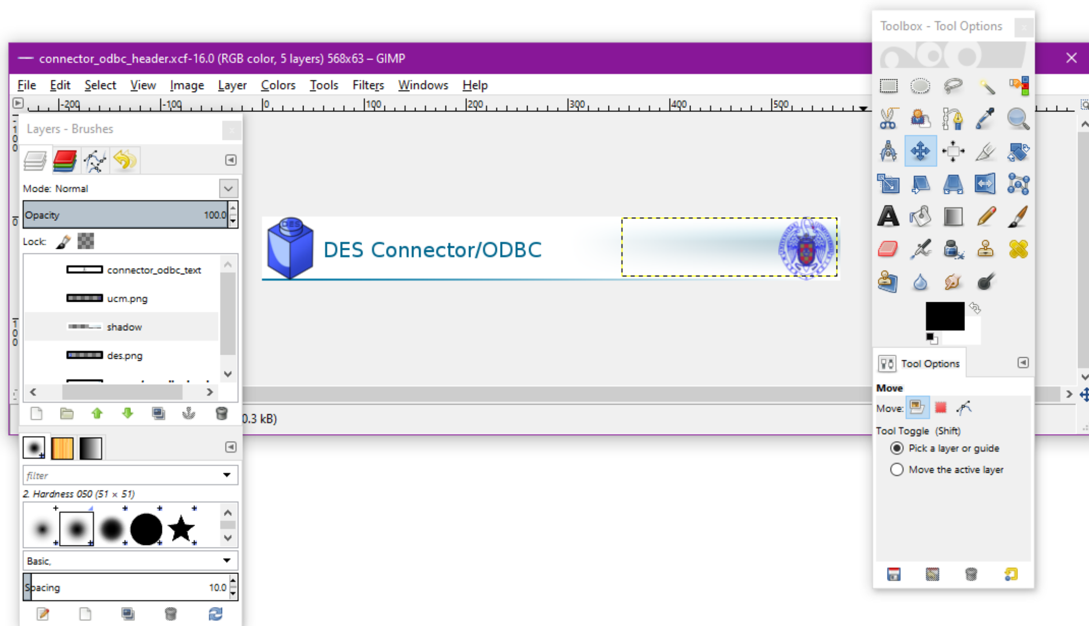


Figura 3.13: Rediseñando el *banner* con GIMP.

3.8.2. Windows

El sistema Windows empleado durante el desarrollo fue Microsoft Windows 10 22H2.

La herramienta principal de desarrollo del conector fue Visual Studio Community 2022. MyODBC presentaba un archivo `.sln`, archivo de solución de Visual Studio, que comprendía dentro de sí todos los proyectos que hemos modificado: versiones ANSI y Unicode del conector, biblioteca de configuración, instalador, tests, y algunas utilidades. Por tanto, el desarrollo de toda la *suite* de DESODBC se realizó principalmente desde Visual Studio.

3.8.2.1. Funcionalidades de Visual Studio

Unas funcionalidades específicas de Visual Studio que empleamos y que merece la pena comentar por la gran utilidad que supusieron, fueron las siguientes:

Comando del *Local Windows Debugger* Cuando se depura desde Visual Studio, existe la posibilidad de ejecutar un comando inicial. Este comando se especifica, partiendo del proyecto deseado a depurar, en *Configuration Properties* → *Debugging* → *Command*. El comando que hemos empleado es el de introducir la ruta de un ejecutable, de modo que este se inicie. Cuando el ejecutable interactúa con nuestras bibliotecas compiladas en modo *debug*, se activan los *breakpoints* necesarios en el

código del conector. El principal ejecutable que empleamos para estos menesteres fue *ODBC Test*, que ahora comentaremos.

Enlazar a un proceso Algunas veces no era posible depurar desde el comando por la naturaleza de los ejecutables: por ejemplo, los ejecutables de línea de comandos. Esto era un problema cuando se pretendía ejecutar programas de prueba que interaccionaran con nuestro conector.

El problema se solucionaba ejecutando manualmente el ejecutable deseado, y dentro del proyecto del conector en Visual Studio, enlazar el conector al ejecutable por medio de la funcionalidad *Attach to Process*. Desde ahí se puede buscar nuestro ejecutable entre la lista de PIDs activos y enlazarlo al mismo. A partir de ahí se puede depurar por *breakpoints* y derivados, tal y como hemos descrito antes.

Dialog Editor Los *Resource Editors*[27] son herramientas de Visual Studio que permiten editar con la comodidad de una interfaz gráfica los archivos *resource script* (`.rc`) como el que define la GUI de nuestra biblioteca de configuración, que en este caso es un `Dialog` y por tanto usamos la herramienta *Dialog Editor*[28]. En la figura 3.14 podemos ver el uso de esta herramienta para diseñar la interfaz gráfica de la biblioteca de configuración.

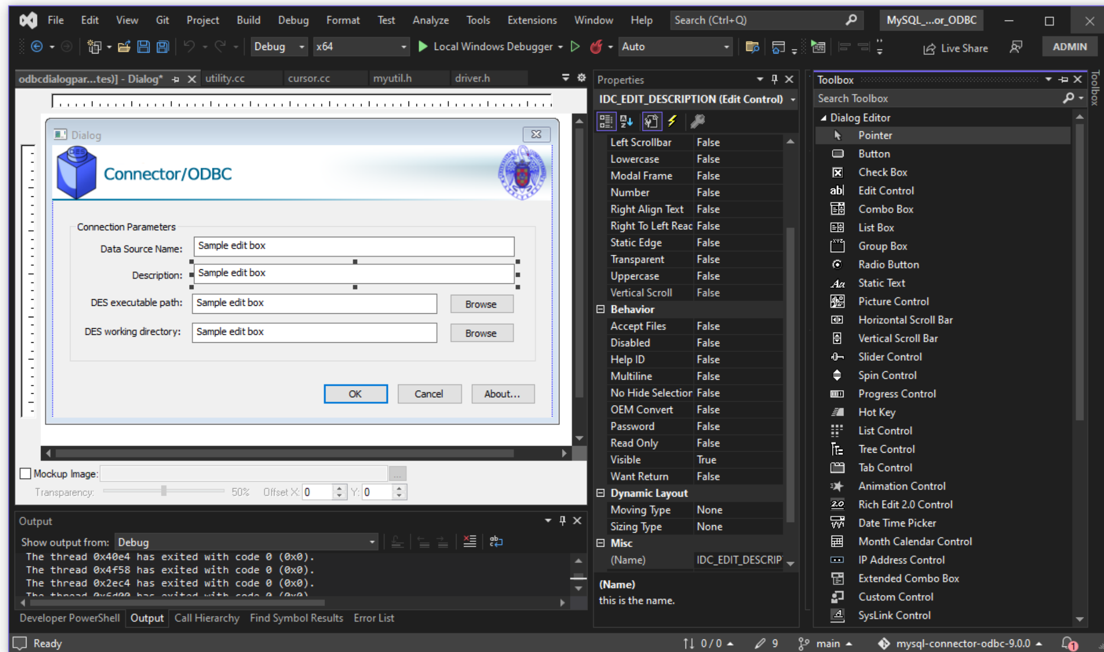


Figura 3.14: Uso del *Dialog Editor* para la biblioteca de configuración.

[27]https://learn.microsoft.com/en-us/cpp/windows/resource-editors?view=msvc-170
[28]https://learn.microsoft.com/en-us/cpp/windows/dialog-editor?view=msvc-170

3.8.2.2. Software ajeno a Visual Studio

ODBC Test El software principal que hemos utilizado para hacer depuración es ODBC Test 02.70[29], una sencilla aplicación de interfaz gráfica que permite entre otras cosas ejecutar las funciones ODBC API sobre un conector que se especifique (véase la figura 3.15).

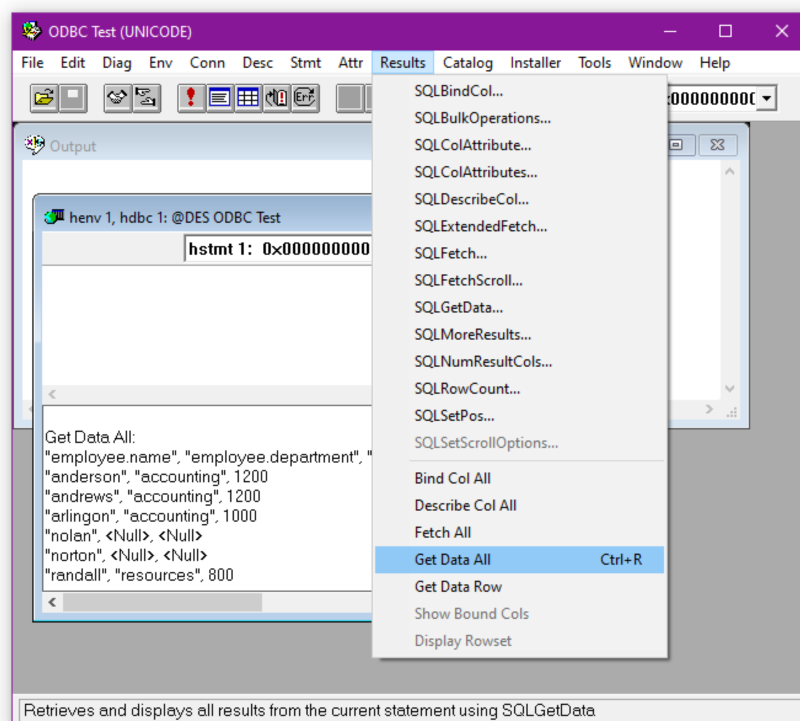


Figura 3.15: Uso de la funcionalidad «*Get Data All*» en ODBC Test, que después de la ejecución de un comando, llama a la sucesión de funciones API pertinentes para mostrar formateado el conjunto de resultados.

ODBC Data Source Administrator Introducido en la figura 3.2, este software nos proporcionaba una interfaz gráfica para añadir, eliminar y modificar fuentes de datos y conectores ODBC.

Registry Editor El software *Registry Editor* viene integrado en Windows, y proporciona una interfaz gráfica para modificar cómodamente las entradas del registro de Windows. Esto nos permite de forma directa a los ficheros `odbc.ini` y `odbcinst.ini` sin pasar por el *ODBC Data Source Administrator*, de modo que entre otras cosas podemos añadir nuevas entradas *key-value* que no sean contempladas por la interfaz de la biblioteca de configuración, permitiéndonos hacer ciertas pruebas.

[29]`https://learn.microsoft.com/en-us/sql/odbc/odbc-test?view=sql-server-ver16`

Dependency Walker En el proceso de compilación de MyODBC en los primeros pasos del desarrollo, al deber compilar la biblioteca `mysqlclient.lib` en modo `Debug` y lidiar con problemas de dependencias con otras bibliotecas, tuvimos que recurrir al software Dependency Walker 2.2[30] para estudiar el árbol de dependencias que se daba entre determinadas bibliotecas.

AccessChk AccessChk[31] es una herramienta de la suite *Sysinternals* que permite acceder en general a los recursos internos del sistema operativo. Tuvimos que hacer uso de esta utilidad (v6.15) para ver en tiempo real qué archivos IPC (memoria compartida, tuberías) existían en memoria. Por ejemplo, para mostrar los recursos vivos en memoria que coincidan parcial o totalmente en nombre con `DESODBC_SHMEM`, se ejecuta en un *shell*:

```
accesschk.exe -osv | findstr DESODBC_SHMEM
```

Esto nos permitió no solo depurar sobre cómo nuestro conector lidiaba con estos recursos, sino también comprender cómo Windows los gestiona, como ya explicamos en el apartado de desconexión.

Process Explorer Este software ya se introdujo en el apartado 3.5.2.2, cuando tratamos la cuestión de las tuberías con el proceso DES. Hemos utilizado la versión v17.06. Nos sirvió para constatar que no era viable acceder a los manejadores entrada y salida que el proceso DES expone al exterior.

Event Viewer *Event Viewer* es un software integrado en Windows y que permite observar eventos lanzados por aplicaciones. Con eventos no nos referimos a los objetos de evento empleados para describir los procesos IPC en Windows, sino a determinados mensajes internos generados mediante las funciones API de Windows `RegisterEventSource` y `ReportEvent` (`winbase.h`). Con *Event Viewer* podemos ver desde una visión global del sistema qué mensajes se han lanzado, desde qué aplicaciones y en qué instantes. Esto era necesario cuando no podíamos depurar con las otras herramientas mencionadas por diversos factores.

Utilidades de shell Se empleó el software PowerShell como *shell* para ejecutar múltiples de las utilidades de sistema en nuestro desarrollo. Una de ellas, por ejemplo, fue para comprobar el estado de los procesos dado su PID (por ejemplo, `Get-Process -Id 11332`). Esto nos permitió saber cuándo y cómo los procesos DES se destruían en los procesos de desconexión.

3.8.3. Unix

Los sistemas operativos basados en Unix empleados para el desarrollo fueron Ubuntu 22.04.5 LTS (Jammy Jellyfish) y macOS Big Sur 11.0.1.

[30] `https://www.dependencywalker.com/`

[31] `https://learn.microsoft.com/es-es/sysinternals/downloads/accesschk`

Visual Studio Code Visual Studio Code es un editor de código fuente, de código abierto, desarrollado por Microsoft. Lo hemos empleado para trabajar sobre el código fuente de nuestro conector y biblioteca de configuración en GNU/Linux y macOS.

Glade (GNU/Linux) Como ya comentamos en el apartado del desarrollo de la biblioteca de configuración, el software Glade[32] nos permitió diseñar la biblioteca de configuración de las versiones GTK. Utilizamos las distintas versiones Glade 3.8.0 para GTK-2 y Glade 3.22.2 para realizar la versión de GTK-3, por una cuestión de incompatibilidades. Puede verse en la figura 3.16 el uso de esta herramienta para diseñar la interfaz gráfica de las bibliotecas de configuración.

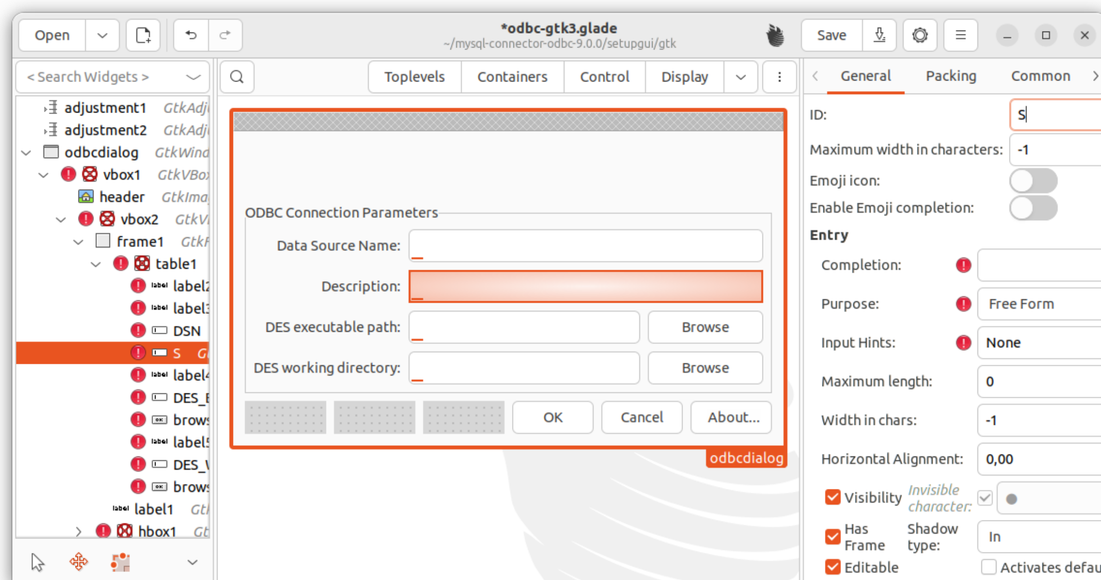


Figura 3.16: Glade 3.22.2 para desarrollar nuestra biblioteca de configuración para GTK-3.

unixodbc-gui-qt (GNU/Linux) Este software[33] comprende una *suite* de herramientas auxiliares a unixODBC que proporcionan una interfaz gráfica basada en Qt. Esto nos permite haber utilizado las herramientas ODBCCreateDataSourceQ5 y ODBCManageDataSourcesQ5 para crear, modificar y eliminar fuentes de datos y conectores de forma cómoda.

***iODBC Data Source Administrator* (macOS)** Se trata de un gestor de fuentes de datos y de conectores, por interfaz gráfica, prácticamente idéntico al *ODBC Data Source Administrator* de Windows. Opera bajo iODBC, solución para ODBC exclusiva para macOS (aunque en macOS también podamos usar unixODBC, solo que no podremos hacer uso de esta utilidad concreta).

[32] `https://glade.gnome.org/`
[33] `https://github.com/mazbrili/unixodbc-gui-qt`

Utilidades de *shell* Pudimos ejecutar múltiples de las utilidades Unix en un *shell* para el desarrollo del conector. Algunas utilidades importantes y poco comunes que empleamos fueron las siguientes:

- `nm`: ver los símbolos contenidos en una biblioteca. Esto es útil para ver a qué funciones se pueden referenciar en una biblioteca ya compilada. Utilizamos esto para errores relacionados con funciones a las que otros software llamaban a nuestra biblioteca de configuración.

- `ipcs`: nos permite lidiar manualmente con los archivos de memoria compartida en RAM como los que creamos con `shmget`. Esto fue útil para depurar.

Capítulo 4

Conclusiones y trabajo futuro

4.1. Conclusiones

Hemos concluido nuestro proyecto habiendo cumplido sobradamente con todos los objetivos propuestos en el apartado 1.2. Así, a partir del conector ODBC de MySQL hemos desarrollado un conector ODBC para DES para todos los sistemas operativos soportados en DES: Windows, GNU/Linux y macOS, ofreciendo además la posibilidad de elegir trabajar en codificación ANSI o Unicode (véase la sección 3.5.7). Además, hemos podido ofrecer por añadidura un instalador en todos los sistemas operativos objetivo, y bibliotecas de configuración no solo en Windows sino también en GNU/Linux (en este sistema, con las consideraciones que se tomaron en la sección 3.4).

No obstante, hemos de reconocer que pese a nuestros esfuerzos existe una cantidad no desdeñable de *bugs* al tratar con el conector dentro de toda la gama de aplicaciones en las que se puede usar. Esto se debe a la gran complejidad de este *software*, con versiones en distintos sistemas operativos, y la variedad de convenciones que las aplicaciones adoptan para tratar con el conector y para enviarle según qué consultas. Nos propondremos como trabajos futuros tomarnos el tiempo necesario para solucionarlos.

Describimos en detalle en la figura 4.1 la lista de las funciones ODBC API y su estado en el producto final.

4.2. Trabajo futuro

A lo largo de este trabajo hemos descrito una serie de problemas y retos encontrados cuya resolución excede los límites de este trabajo de fin de grado, bien por su complejidad o bien por la cantidad de tiempo requerida. Los recopilamos en este apartado y añadimos otros nuevos tras una reflexión, y son:

1. Implementación de cursores dinámicos.

2. Implementación de enteros de tamaño ilimitado, en contraste con los enteros de 64 bits en la versión actual.

3. Implementación de una biblioteca de configuración para la versión de macOS. Estudiar cómo conseguir que las bibliotecas de configuración GTK en GNU/-Linux sean compatibles con la solución GUI externa que unixODBC refiere (`unixodbc-gui-qt`).

4. Aunque no sea lo estándar en un producto de estas características, considerar la introducción de traducciones para los mensajes de texto en la *suite* más allá del inglés.

5. Estudiar la viabilidad de rediseñar la arquitectura con técnicas avanzadas que eviten el uso de tuberías con nombre y semáforos con nombre en determinadas plataformas.

6. Probar exhaustivamente las aplicaciones que hacen uso de ODBC para depurar los *bugs* que DESODBC presenta.

SQLAllocConnect (N/A)
SQLAllocEnv (N/A)
SQLAllocHandle (R)
SQLAllocStmt (N/A)
SQLBindCol (R)
SQLBindParameter (R)
SQLBrowseConnect (X)
SQLBulkOperations (M)
SQLCancel (X)
SQLCancelHandle (X)
SQLCloseCursor (R)
SQLColAttribute (R)
SQLColAttributes (N/A)
SQLColumnPrivileges (X)
SQLColumns (M)
SQLCompleteAsync (X)
SQLConnect (M)
SQLCopyDesc (R)
SQLDataSources (N/A)
SQLDescribeCol (R)
SQLDescribeParam (M)
SQLDisconnect (M)
SQLDriverConnect (M)
SQLDrivers (N/A)
SQLEndTran (X)
SQLError (N/A)
SQLExecDirect (M)
SQLExecute (M)
SQLExtendedFetch (R)
SQLFetch (R)
SQLFetchScroll (R)
SQLForeignKeys (M)
SQLFreeConnect (N/A)
SQLFreeEnv (N/A)
SQLFreeHandle (R)
SQLFreeStmt (M)
SQLGetConnectAttr (M)
SQLGetConnectOption (N/A)
SQLGetCursorName (M)
SQLGetData (R)
SQLGetDescField (M)
SQLGetDescRec (R)
SQLGetDiagField (M)
SQLGetDiagRec (R)
SQLGetEnvAttr (M)
SQLGetFunctions (M)
SQLGetInfo (M)
SQLGetStmtAttr (M)
SQLGetStmtOption (N/A)
SQLGetTypeInfo (M)
SQLMoreResults (M)
SQLNativeSql (R)
SQLNumParams (R)
SQLNumResultCols (R)
SQLParamData (X)
SQLPrepare (R)
SQLPrimaryKeys (M)
SQLProcedureColumns (X)
SQLProcedures (X)
SQLPutData (X)
SQLRowCount (R)
SQLSetConnectAttr (M)
SQLSetConnectOption (N/A)
SQLSetCursorName (R)
SQLSetDescField (M)
SQLSetDescRec (R)
SQLSetEnvAttr (M)
SQLSetParam (N/A)
SQLSetPos (M)
SQLSetStmtAttr (M)
SQLSetStmtOption (N/A)
SQLSpecialColumns (M)
SQLStatistics (M)
SQLTablePrivileges (X)
SQLTables (M)
SQLTransact (N/A)

N/A: no procede implementar (deprecada o función de Driver Manager).
M: modificado de MyODBC para DESODBC.
R: reutilizado por completo de MyODBC.
X: no implementado por las características de DES.

Figura 4.1: Estado de las funciones ODBC API en DESODBC.

Introduction

This chapter contains the translation of Chapter 1.

4.3. Preliminaries and motivation

This work consists of an implementation of an ODBC Driver for the DBMS (Data Base Management System) Datalog Educational System (DES).

4.3.1. What is an ODBC?

ODBC (Open Database Connectivity) is «*a standard application programming interface (API) for accessing data in both relational and nonrelational database management systems (DBMSs)*» [1, pp. 3-4], of last version 3.8 (2009) [2]. This standard was developed in the early 90s by Microsoft and Simba Technologies, and solved a previously impracticable problem: being able to access multiple databases through the SQL standard from a single application, without having to program specific code in there for these connections [1, p. 3].

Thus, a DBMS could have its own ODBC driver that supported ODBC API function calls for communication with one of the DBMS's data sources.

Therefore, and application wishing to connect to such data source would only need to communicate with an ODBC Driver Manager (for instance, in Windows, `System32/odbc32.dll`) that acted as an intermediary between the application and the connector. In this way, communications between the application and the desired data source would take place without needing any extra programming; that is, it would be agnostic to the driver's implementation and the nature of the data source or its associated DBMS.

4.3.2. Datalog Educational System (DES)

Datalog Educational System (DES) is a deductive database management system which is open-source, free, cross-platform, and published originally in 2004 by Fernando Sáenz-Pérez, Associate Professor / *Profesor Titular* of Universidad Complutense de Madrid [3]. DES, among many of its characteristics, supports queries

in the languages Datalog, SQL, Relational Algebra, Tuple Relational Calculus and Domain Relational Calculus [4].

The developer, who has also been responsible for its maintenance until today, had as his main objective the use of DES in the educational field for students learning databases, aiming to teach intuitively and interactively the concepts behind a deductive database, as well as teaching the supported languages for queries that we have listed [7, p. 10]. However, its use spread to universities all around the world, having as a collateral effect even its use in research in different fields [5].

The author of this work was one of these students who were introduced to this system, with the developer himself as his professor in the course *Bases de Datos* (Databases) in third year of the *doble grado en Ingeniería Informática y Matemáticas* ("double degree" in Computer Science and Mathematics); this professor is also the tutor of this work.

4.3.3. Proposed problem

The director of this work described the problem in the following lines[1]:

> *«ODBC (Open Database Connectivity) is a standard API for accessing data sources, mainly relational databases, developed by Microsoft and Simba. Currently there are ODBC drivers for the majority of databases (MySQL, PostgreSQL, DB2...) and event to other sources such as Excel and CSV files. In this project a new ODBC driver will be developed for the deductive database system DES (`http://des.sourceforge.net`), that supports various query languages on relational data structures. The usual programming language used for developing these drivers is C, and one can start from the Microsoft docs and already done open-source implementations such as those for PostgreSQL and MySQL. Given that DES does not support concurrent queries, it will be necessary to implement its management in the proper driver. Depending on the number of students, connectors can be developed not only for Windows, but also for Linux and macOS.»* [6]

Indeed, even though DES is able to connect to a ODBC Driver for communicating with other DBMSs, DES lacks its own ODBC driver. That is, there is no driver through which a ODBC Driver Manager could communicate with DES, so that external applications could connect to specific databases of this DBMS.

4.4. Objectives

Once we have present the proposed problem, the objective of this work is to develop an ODBC Driver, understanding that its fundamental characteristics will be: (1) supporting calls from the ODBC Driver Manager, which demands some information; (2) access to the DES databases through its own already implemented

[1] I have translated it into English from the Spanish original.

API (Textual API or TAPI, as we will see), and (3) return the requested information to the ODBC Driver Manager. For this, we will start from the source code of MyODBC, the ODBC Driver of MySQL.

We will implement a ODBC Driver for the last version of ODBC (3.8) for the last version for DES (V6.7). We will propose as the concrete minimal objective offering a ODBC Driver for DES in all the operating systems that DES supports: Windows, GNU/Linux and macOS.

Given that we start from the already implemented MyODBC driver for the DBMS MySQL, we will also propose as optional objectives implement all the functionalities that its suite offers, namely:

- Installer executable capable of:
  - Registering the ODBC driver in the system.
  - Adding and updating data sources.
- GUI (Graphical User Interface) setup in Windows for the parameters of a data source associated to the ODBC driver[2].
- The ODBC driver itself, with versions supporting:
  - Operating systems Windows, GNU/Linux with unixODBC, macOS with unixODBC and iODBC[3].
  - ANSI and Unicode formatting[4].

In synthesis, we will satisfy the main objective but we will also strive to achieve these last optional objectives. With this, we will accomplish more than enough the director's proposed objectives in the presentation of the problem (section 4.3.3).

With the implementation of this software, we will have revised all the software concepts that are taught in the current study guidelines: we deal with object-oriented programming, databases, data structures, algorithms, concurrency policies,

[2]MyODBC in its source code provides means for compiling GUI setup libraries for GNU/Linux. These libraries are not distributed in the official versions of MyODBC, so as a result we will not include providing setup libraries for GNU/Linux as a requisite. However, we have worked on these extra-official libraries and we had included them in the final work with some considerations (see section 3.4).

[3]In reality, the MyODBC driver supports other *Unix-like* systems such as Oracle Solaris. DES can be used in this last operating system if both DES and its dependencies SWI-Prolog/SICStus are compiled from their respective sources. DES does not present an official version for Oracle Solaris: consequently, we then restrict ourselves to the systems in which DES offers support: Windows, GNU/Linux, macOS.

However, we believe that, excepting any minor issues that may need to be resolved, this driver can be compiled and work in Oracle Solaris. The reason why is that, as we will see, our implementation is practically agnostic to the Unix-like systems in which it is destined, with some minor exceptions regarding differences relative to POSIX functions.

[4]In the section 3.5.7 we will study what we are refering to.

IPC mechanisms, GUI designing (and its respective criteria of proper interactive systems designing), correctness, internal aspects of operating systems, use of software development tools, etc.

4.5. Work plan

1. Research phase (*July 2024 - August 2024, January 2025 - March 2025*). The questions we will need to answer are:

   - What is ODBC.
   - What is a ODBC Driver.
   - How is the usage flow of an ODBC Driver and the agents implied in it.
   - Which DES functionalities we will need to use.
   - What functionalities MyODBC has.
   - What MyODBC means by ANSI and Unicode.
   - How an ODBC connection works low-level in Windows and *Unix-like* systems.
   - What functionalities of Windows API and POSIX can we use.

   Part of these questions will be answered and researched throughout our development phase while we find unpredictable challenges. This is why the research and development phases are broad, general tasks that intertwine and overlap in time.

2. Development phase (*July 2024 - August 2024, January 2025 - March 2025*). The working flux we have considered best is the following:

   a) Importing and compiling the MyODBC source code in Windows.
   b) Parallel process of adapting the driver to the DES needs, and removal of MySQL-related external libraries that we do not need.
   c) Adaptation to the *Unix-like* systems GNU/Linux and macOS.

3. Testing and debugging phase (*April 2025 - May 2025*).

   - Testing with typical use cases as well as «pathological» ones so as to debug possible concept errors, bugs, and security flaws.
   - Debugging and refinement of the driver beyond that of the development phase.

4. Writing of the final report and documenting the source code (*April 2025 - May 2025*).

The figure 4.2 shows the Gantt chart that portrays the flow of the concrete and broken-down tasks.

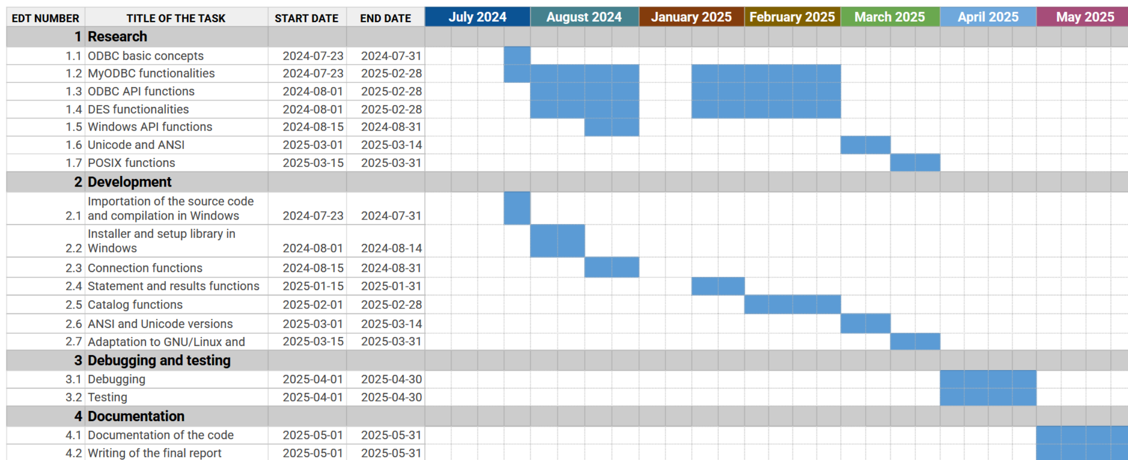


Figure 4.2: Gantt chart associated with our work plan.

4.6. Final product

The author of this work would end up doing not only the Windows implementation but also the GNU/Linux and macOS ones. In the director's proposed objectives it is explained that the scenery of exporting to different operating systems would apply in the case of multiple students. We explain at this point that, as we will see, the code of the functionalities themselves of the driver is directly exportable for all the compilers and operating systems; the only thing that we will have to take into account for each system, is the aspect of connecting to a DES process and the implied Inter-Process Communications (IPCs) implied, as well as some aspects regarding the assistant tools (setup and installer, as we will see). This is the reason why the author of this work considered that the extra work to do was not enough for requiring a team.

Our final product is an ODBC Driver called DESODBC and its GitHub repository where it is stored can be visited at the following link: `https://github.com/segarc21/des-connector-odbc`.

4.7. Organization of the document

This document describes in the chapter 3 the development of the research and implementation phases in parallel for didactic and exhibition purposes. Conclusions in English will be described in the section 4.7.

Conclusions and future work

4.8. Conclusions

We have finished our project having accomplished more than enough with all the objectives proposed in the section 4.4. So, starting from the ODBC Driver of MySQL, we have developed an ODBC Driver for DES for all the operating systems supported in DES: Windows, GNU/Linux and macOS, offering as well the possibility of choosing to work in ANSI or Unicode encoding (see section 3.5.7). Besides, we have been able to offer in addition an installer in every objective operating system, and a setup library not only in Windows but also in GNU/Linux (in this system, with the considerations we took in section 3.4).

However, we must acknowledge that despite our efforts, there is a not negligible number of bugs when dealing with the driver accross the wide variety of applications where it can be used. This is due to the high complexity of this software, with versions in different operating systems, and the variety of conventions that applications adopt when treating an ODBC Driver and sending them certain queries. We aim to allocate sufficient time in future works to solve them.

Figure 4.3 shows the list of the ODBC API functions and their state in the final product.

4.9. Future work

Throughout this work we have described a series of found problems and challenges whose resolutions exceed the limits of this work, either because of its complexity or the amount of time required. We list them as well as some new others in this section, and they are:

1. Implementation of dynamic cursors.
2. Implementation of unlimited size integers, in contrast with the current limited 64 bit integers.
3. Implementation of a setup library for the macOS version. Study how to achieve full compability between the GTK setup libraries in GNU/Linux and the external GUI solution unixODBC refers to (`unixodbc-gui-qt`).

4. Though it is not the standard for a product of these characteristics, considering to support translations for the text messages throughout the suite beyond English.

5. Studying the viability of redesigning the architecture with advanced techniques that avoid the use of named pipes and named semaphores in certain platforms.

6. Thoroughly test the applications using ODBC to fix the bugs that the connector presents.

SQLAllocConnect (N/A)
SQLAllocEnv (N/A)
SQLAllocHandle (R)
SQLAllocStmt (N/A)
SQLBindCol (R)
SQLBindParameter (R)
SQLBrowseConnect (X)
SQLBulkOperations (M)
SQLCancel (X)
SQLCancelHandle (X)
SQLCloseCursor (R)
SQLColAttribute (R)
SQLColAttributes (N/A)
SQLColumnPrivileges (X)
SQLColumns (M)
SQLCompleteAsync (X)
SQLConnect (M)
SQLCopyDesc (R)
SQLDataSources (N/A)
SQLDescribeCol (R)
SQLDescribeParam (M)
SQLDisconnect (M)
SQLDriverConnect (M)
SQLDrivers (N/A)
SQLEndTran (X)
SQLError (N/A)
SQLExecDirect (M)
SQLExecute (M)
SQLExtendedFetch (R)
SQLFetch (R)
SQLFetchScroll (R)
SQLForeignKeys (M)
SQLFreeConnect (N/A)
SQLFreeEnv (N/A)
SQLFreeHandle (R)
SQLFreeStmt (M)
SQLGetConnectAttr (M)
SQLGetConnectOption (N/A)
SQLGetCursorName (M)
SQLGetData (R)
SQLGetDescField (M)
SQLGetDescRec (R)
SQLGetDiagField (M)
SQLGetDiagRec (R)
SQLGetEnvAttr (M)
SQLGetFunctions (M)
SQLGetInfo (M)
SQLGetStmtAttr (M)
SQLGetStmtOption (N/A)
SQLGetTypeInfo (M)
SQLMoreResults (M)
SQLNativeSql (R)
SQLNumParams (R)
SQLNumResultCols (R)
SQLParamData (X)
SQLPrepare (R)
SQLPrimaryKeys (M)
SQLProcedureColumns (X)
SQLProcedures (X)
SQLPutData (X)
SQLRowCount (R)
SQLSetConnectAttr (M)
SQLSetConnectOption (N/A)
SQLSetCursorName (R)
SQLSetDescField (M)
SQLSetDescRec (R)
SQLSetEnvAttr (M)
SQLSetParam (N/A)
SQLSetPos (M)
SQLSetStmtAttr (M)
SQLSetStmtOption (N/A)
SQLSpecialColumns (M)
SQLStatistics (M)
SQLTablePrivileges (X)
SQLTables (M)
SQLTransact (N/A)

N/A: not appliable (deprecated or a Driver Manager function).
M: modified from MyODBC for DESODBC.
R: reused completely from MyODBC.
X: not implemented due to DES' characteristics.

Figure 4.3: State of the ODBC API functions in DESOBDC.

Bibliografía

[1] Geiger, K. *Inside ODBC*. Microsoft Press, 1995.

[2] Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *ODBC programmer's reference - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/odbc-programmer-s-reference?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[3] Sáenz-Pérez, F. *Datalog Educational System V1.1*, User's Manual. Technical Report 139-04, Facultad de Informática, Universidad Complutense de Madrid. 2024.

[4] Sáenz-Pérez, F. *Key Features*. Datalog Educational System. n.d. Disponible en `https://www.fdi.ucm.es/profesor/fernan/des/html/key_features.html` (último acceso: 17 de mayo de 2025).

[5] Sáenz-Pérez, F. *What's DES for?*. Datalog Educational System. n.d. Disponible en `https://www.fdi.ucm.es/profesor/fernan/des/html/what_s_des_for_.html` (último acceso: 17 de mayo de 2025).

[6] Sáenz-Pérez, F. *TFG. Propuestas de proyectos de Trabajo Fin de Grado (TFG) curso 2024-2025*. Fernando Sáenz Pérez. n.d. Disponible en `https://www.fdi.ucm.es/profesor/fernan/fsp/html/tfg.html` (último acceso: 17 de mayo de 2025).

[7] Sáenz-Pérez, F. *Datalog Educational System V6.7*, User's Manual. 04 de septiembre de 2021. Disponible en `https://www.fdi.ucm.es/profesor/fernan/des/html/manual/manualDES.html` (último acceso: 17 de mayo de 2025).

[8] Batchelor, D., Cai, S., Coulter, D., Satran, M., Sharkey, K. y White, S. *About dynamic-link libraries - Win32 apps*. Microsoft Learn. 07 de enero de 2021. Disponible en `https://learn.microsoft.com/en-us/windows/win32/dlls/about-dynamic-link-libraries` (último acceso: 17 de mayo de 2025).

[9] MySQL. *4.4 Building Connector/ODBC from a source distribution on Windows*. MySQL Connector/ODBC Developer Guide. n.d.

Disponible en https://dev.mysql.com/doc/connector-odbc/en/connector-odbc-installation-source-windows.html (último acceso: 17 de mayo de 2025).

[10] Cai, S., Satran, M. y Walker, J. *Resource Compiler - Win32 apps*. Microsoft Learn. 23 de agosto de 2019. Disponible en https://learn.microsoft.com/es-es/windows/win32/menurc/resource-compiler (último acceso: 17 de mayo de 2025).

[11] unixODBC. unixODBC-Gui. n.d. Disponible en https://www.unixodbc.org/gui/ (último acceso: 17 de mayo de 2025).

[12] Sanders, R. E. *ODBC 3.5 Developer's Guide*. McGraw-Hill Professional, 1998.

[13] Engel, D. and Guyer, C. and Masha, T. and Rabeler, C. and Roth, J. and West R. *Handles - ODBC API reference*. Microsoft Learn. 18 de octubre de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/handles?view=sql-server-ver17 (último acceso: 17 de mayo de 2025).

[14] Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *Buffers - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/buffers?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[15] Engel, D. and Guyer, C. and Masha, T. and Rabeler, C. and Roth, J. and West R. *Binding Parameters ODBC - ODBC API reference*. Microsoft Learn. 18 de octubre de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/binding-parameters-odbc?view=sql-server-ver17 (último acceso: 17 de mayo de 2025).

[16] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLGetInfo Function - ODBC API reference*. Microsoft Learn. 18 de octubre de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlgetinfo-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[17] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLConnect Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/es-es/sql/odbc/reference/syntax/sqlconnect-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[18] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLDriverConnect Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/

reference/syntax/sqldriverconnect-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[19] ANDREWS, G. R. *Foundations of multithreaded, parallel, and distributed programming*. Wesley, University of Arizona, USA, 2000.

[20] BATCHELOR, D., CAI, S., COULTER, D., JACOBS, M., SATRAN, M., SHARKEY, K. y WHITE, S. *Named Pipes - Win32 apps*. Microsoft Learn. 01 de julio de 2021. Disponible en https://learn.microsoft.com/en-us/windows/win32/ipc/named-pipes (último acceso: 17 de mayo de 2025).

[21] ANDERSSON, H., BATCHELOR, D., BRIDGE, K., CAI, S., COULTER, D., DOTTERRER, K., KATSIOPIS, N., SATRAN, M. y SHARKEY, K. *Resource Compiler - Win32 apps*. Microsoft Learn. 07 de julio de 2022. Disponible en https://learn.microsoft.com/es-es/windows/win32/menurc/resource-compiler (último acceso: 17 de mayo de 2025).

[22] BATCHELOR, D., CAI, S., COULTER, D., SATRAN, M., SHARKEY, K., WHITE, S. ET AL. *Event Objects (Synchronization) - Win32 apps*. Microsoft Learn. 30 de enero de 2021. Disponible en https://learn.microsoft.com/en-us/windows/win32/sync/event-objects (último acceso: 17 de mayo de 2025).

[23] ENGEL, D., GUYER, C., RABELER, C., ROTH, J., WEST R. ET AL. *SQLDisconnect Function - ODBC API reference*. Microsoft Learn. 17 de diciembre de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqldisconnect-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[24] MICROSOFT. *CloseHandle Function - (handleapi.h)*. Microsoft Learn. 13 de octubre de 2021. Disponible en https://learn.microsoft.com/es-es/windows/win32/api/handleapi/nf-handleapi-closehandle (último acceso: 17 de mayo de 2025).

[25] ENGEL, D., GUYER, C., MASHA, T., RABELER, C., ROTH, J. y R., W. *Driver Manager's Role in the Connection Process - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/driver-manager-s-role-in-the-connection-process?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[26] UNIXODBC. *Closing a connection*. ODBC Programming Tutorial. n.d. Disponible en https://www.unixodbc.org/doc/ProgrammerManual/Tutorial/close.html (último acceso: 17 de mayo de 2025).

[27] MICROSOFT. *OBJECT_ATTRIBUTES structure (ntdef.h) - Win32*. Microsoft Learn. 09 de agosto de 2023. Disponible en https://learn.microsoft.com/en-us/windows/win32/api/ntdef/ns-ntdef-_object_attributes (último acceso: 17 de mayo de 2025).

[28] Cai, S. and Casey C. and Robertson, C. and Sharkey, K. and Whitney T. et al. *atexit - C++, C, and Assembler*. Microsoft Learn. 26 de octubre de 2022. Disponible en `https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/atexit?view=msvc-170` (último acceso: 17 de mayo de 2025).

[29] Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *Scrollable Cursor Types - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/scrollable-cursor-types?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[30] Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *Block cursors - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/block-cursors?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[31] Potapchenko, A., Tsvigun, V., Cisar P., Starkey J. et al. *FireBird ODBC/JDBC Driver 2.0 Manual*. 30 de agosto de 2020. Disponible en `https://www.firebirdsql.org/file/documentation/html/en/refdocs/fbodbc20/firebird-odbc-driver-20-manual.html` (último acceso: 17 de mayo de 2025).

[32] Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *Bookmarks (ODBC) - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/bookmarks-odbc?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[33] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLSetPos Function - ODBC API reference*. Microsoft Learn. 01 de febrero de 2025. Disponible en `https://learn.microsoft.com/es-es/sql/odbc/reference/syntax/sqlsetpos-function?view=sql-server-ver17` (último acceso: 17 de mayo de 2025).

[34] MySQL. *15.2.2 DELETE Statement*. MySQL 8.4 Reference Manual. n.d. Disponible en `https://dev.mysql.com/doc/refman/8.4/en/delete.html` (último acceso: 17 de mayo de 2025).

[35] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLBulkOperations Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlbulkoperations-function?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[36] Silberschatz, A., Korth, H. F. y Sudarshan, S. *Database system concepts. Seventh edition*. McGraw-Hill Education, 2019.

[37] MySQL. *8.1.3 Configuring Catalog and Schema Support*. MySQL Connector/ODBC Developer Guide. n.d. Disponible en https://dev.mysql.com/doc/connector-odbc/en/connector-odbc-usagenotes-functionality-catalog-schema.html (último acceso: 17 de mayo de 2025).

[38] Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *Pattern Value Arguments - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/pattern-value-arguments?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[39] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLColumns Function - ODBC API reference*. Microsoft Learn. 01 de febrero de 2025. Disponible en https://learn.microsoft.com/es-es/sql/odbc/reference/syntax/sqlcolumns-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[40] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLColumnPrivileges Function - ODBC API reference*. Microsoft Learn. 01 de febrero de 2025. Disponible en https://learn.microsoft.com/es-es/sql/odbc/reference/syntax/sqlcolumnprivileges-function?view=sql-server-ver17 (último acceso: 17 de mayo de 2025).

[41] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLSpecialColumns Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlspecialcolumns-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[42] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLPrimaryKeys Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlprimarykeys-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[43] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLForeignKeys Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlforeignkeys-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[44] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLProcedures Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlprocedures-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[45] MySQL. *15.1.17 CREATE PROCEDURE and CREATE FUNCTION Statements*. MySQL 8.4 Reference Manual. n.d. Disponible en https://dev.mysql.com/doc/refman/8.4/en/create-procedure.html (último acceso: 17 de mayo de 2025).

[46] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLTables Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/es-es/sql/odbc/reference/syntax/sqltables-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[47] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *C Data Types - ODBC API reference*. Microsoft Learn. 17 de diciembre de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/appendixes/c-data-types?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[48] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQLGetData Function - ODBC API reference*. Microsoft Learn. 17 de diciembre de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlgetdata-function?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[49] Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQL to C Data Conversion Examples - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/appendixes/sql-to-c-data-conversion-examples?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[50] Carlsson M. et al. *SICStus Prolog User's Manual*. RISE Research Institutes of Sweden AB. Marzo de 2025. Disponible en https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf (último acceso: 17 de mayo de 2025).

[51] SWI-Prolog. *SWI-Prolog Reference Manual*. n.d. Disponible en https://www.swi-prolog.org/man/arith.html (último acceso: 17 de mayo de 2025).

[52] Cai, S., Engel, D., Guyer, C., Masha, T., Rabeler, C., Roth, J. y R., W. *C Data Types in ODBC - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/develop-app/c-data-types-in-odbc?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[53] Cai, S., Engel, D., Guyer, C., Rabeler, C., Roth, J., West R. et al. *SQL Data Types - ODBC API reference*. Microsoft Learn. 26 de agosto de 2024. Disponible en https://learn.microsoft.com/en-us/sql/odbc/reference/appendixes/sql-data-types?view=sql-server-ver16 (último acceso: 17 de mayo de 2025).

[54] ENGEL, D., GUYER, C., RABELER, C., ROTH, J., WEST R. ET AL. *SQLStatistics Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlstatistics-function?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[55] ENGEL, D., GUYER, C., RABELER, C., ROTH, J., WEST R. ET AL. *SQLGetDiagRec Function - ODBC API reference*. Microsoft Learn. 25 de junio de 2024. Disponible en `https://learn.microsoft.com/en-us/sql/odbc/reference/syntax/sqlgetdiagrec-function?view=sql-server-ver16` (último acceso: 17 de mayo de 2025).

[56] COMPART AG. *Unicode Character U+24B62*. 2021. Disponible en `https://www.compart.com/en/unicode/U+24B62` (último acceso: 17 de mayo de 2025).

[57] MYSQL. *MySQL Connector/C++ 9.2 Developer Guide*. n.d. Disponible en `https://dev.mysql.com/doc/connector-cpp/9.2/en/` (último acceso: 17 de mayo de 2025).

Apéndice A

Instalación rápida del conector

En este apéndice describimos una pequeña guía de usuario de cómo instalar DESODBC. Los prerrequisitos son:

- Tener localizada una instalación de DES.
- Tener localizados los siguientes ficheros de la *suite* DESODBC: instalador, librería del conector, y para Windows y GNU/Linux una librería de configuración de DESODBC.
- En GNU/Linux, tener instalado unixODBC; en macOS, tener instalado iODBC o unixODBC.

En el repositorio[1] puede hallarse tanto la *Release* con los binarios como esta misma guía más desarrollada en el `README`. Vamos a hablar aquí de una instalación típica.

Haremos uso de nuestra herramienta de línea de comandos `desodbc-installer`, componente dentro de la *suite* DESODBC. Esta herramienta idéntica en todas las plataformas permite instalar (así como modificar y desinstalar) tanto el conector como una fuente de datos en el sistema. Si se llama a `desodbc-installer` sin ningún argumento (o con `-h`), se puede observar una descripción detallada de cómo operar con la herramienta. En la figura A.1 puede observarse una traducción de este pequeño manual de usuario.

Procedemos a introducir una manera rápida y sencilla de registrar el conector y una fuente de datos.

Primero registramos el conector ODBC en el sistema, ejecutando en modo administrador (importante)[2] el comando prototípico que se presenta en la figura A.2. En macOS debemos omitir la subcadena `SETUP` del atributo cadena especificado en la figura (DESODBC no presenta biblioteca de configuración en macOS).

[1]Recordamos que el enlace se presentó en la sección 1.4: `https://github.com/segarc21/des-connector-odbc`.

[2]En Windows, una forma de hacer esto es abrir la shell en modo administrador. En sistemas basados en Unix podemos llamar al comando añadiendo al principio la palabra `sudo`.

Una vez hayamos ejecutado el comando y recibido una salida `Success`, podemos empezar a registrar una fuente de datos sobre la cual trabajar. Para ello, una posibilidad es ejecutar el comando de la figura A.3. En esta figura A.3 se ha especificado implícitamente la opción de crear una fuente de datos de sistema, por lo que se precisan permisos de administrador. Nótese, además, que debemos de especificar como *nombre de la instalación del conector* el que hemos especificado al principio en su instalación.

Se presenta en las figuras A.4, A.5 y A.6 un ejemplo de instalación rápida en los distintos sistemas y que termina creando una fuente de datos de sistema.

Otra alternativa para registrar el conector y crear una fuente de datos es la modificación manual de los ficheros `odbc.ini` y `odbcinst.ini` pertinentes. Se trata de una forma universal de instalación que puede consultarse en manuales ODBC. Como acabamos de proporcionar una alternativa más amigable al usuario, no describiremos cómo llevar a cabo esta instalación manual.

Por último, recordemos que podemos modificar nuestras fuentes de datos de distintas maneras según el sistema, de acuerdo con lo visto en la sección 3.4.

```
> desodbc-installer

Sintaxis
|
|    desodbc-installer <Objeto> <Acción> [Opciones]
|
| Objeto
|
|     -d conector
|     -s fuente de datos
|
| Acción
|
|     -l listar
|     -a añadir (añadir/actualizar para fuentes de datos)
|     -r eliminar
|     -h mostrar esta página de ayuda y salir
|
| Opciones
|
|     -n <nombre>
|     -t <cadena atributo>
|        si se usa para gestionar nombres de fuentes de datos,
|        <attribute string> debe contener opciones ODBC para la
|        ruta ejecutable de DES y la ruta del directorio de
|        trabajo desde el cual DES debe lanzarse.
|     -c0 añade la fuente de datos como usuario y como sistema
|     -c1 añade la fuente de datos como usuario
|     -c2 añade la fuente de datos como sistema (opción por
|         defecto)
|
|     Nótese que añadir una fuente de datos como sistema puede
|     requerir permisos de administrador.
```

Figura A.1: Pequeño manual de usuario de `desodbc-installer` incluido en la herramienta (traducción al castellano).

```
> desodbc-installer -d -a -n "nombre de la instalación del
  conector" -t "DRIVER=ruta de la biblioteca del conector;SETUP=
  ruta de la biblioteca de configuración"
```

-d: conector (*driver*)
-a: añadir
-n: nombre
-t: atributo cadena

Figura A.2: Instalación típica del conector en el sistema. El formateado de esta figura presenta líneas partidas no presentes en el comando real.

```
> desodbc-installer -s -a -n "nombre deseado de la fuente
  de datos del conector" -t
  "DRIVER=nombre de la instalación del conector;
   DESCRIPTION=descripción deseada;
   DES_EXEC=ruta del ejecutable DES;
   DES_WORKING_DIR=ruta del directorio de trabajo deseado"
```

-s: fuente de datos (*data source*)
-a: añadir
-n: nombre
-t: atributo cadena

Figura A.3: Ejemplo de instalación de una fuente de datos. El formateado de esta figura presenta líneas partidas no presentes en el comando real.

```
> desodbc-installer.exe -d -a -n "DESODBC Unicode Driver" -t
  "DRIVER=C:\Users\sergio\DESODBC\desodbcw.dll;
  SETUP=C:\Users\sergio\DESODBC\desodbcS.dll"
Success: Usage count is 1

> desodbc-installer.exe -s -a -n "DESODBC Data Source Test" -t
  "DRIVER=DESODBC Unicode Driver;DESCRIPTION=Test;
  DES_EXEC=C:\des\des.exe;DES_WORKING_DIR=C:\des"
Success
```

Figura A.4: Ejemplo de instalación del conector en Windows (debe abrirse previamente la consola en modo administrador). En caso de error, consultar archivo `README` del repositorio. El formateado de esta figura presenta líneas partidas no presentes en el comando real.

```
> sudo ./desodbc-installer -d -a -n "DESODBC Unicode Driver"
  -t "DRIVER=/home/sergio/DESODBC/libdesodbcw.so;
  SETUP=/home/sergio/DESODBC/libdesodbcS.so"
Success: Usage count is 1

> sudo ./desodbc-installer -s -a -n "DESODBC Data Source Test"
  -t "DRIVER=DESODBC Unicode Driver;DESCRIPTION=Test;
  DES_EXEC=/home/sergio/des/des;DES_WORKING_DIR=/home/sergio/des"
Success
```

Figura A.5: Ejemplo de instalación del conector en GNU/Linux (modo administrador). En caso de error, consultar archivo `README` del repositorio. El formateado de esta figura presenta líneas partidas no presentes en el comando real.

```
> sudo ./desodbc-installer -d -a -n "DESODBC Unicode Driver"
  -t "DRIVER=/Users/sergio/DESODBC/libdesodbcw.so;
  SETUP=/Users/sergio/DESODBC/libdesodbcS.so"
Success: Usage count is 1

> sudo ./desodbc-installer -s -a -n "DESODBC Data Source Test"
  -t "DRIVER=DESODBC Unicode Driver;DESCRIPTION=Test;
  DES_EXEC=/Users/sergio/des/des;DES_WORKING_DIR=/Users/sergio/des"
Success
```

Figura A.6: Ejemplo de instalación del conector en macOS (modo administrador). En caso de fallo, consultar archivo `README` del repositorio. El formateado de esta figura presenta líneas partidas no presentes en el comando real.