Proyecto

Aplicaciones Distribuidas en Internet/Intranets: de los Sockets a los Objetos Distribuidos*

Alumno
Diego Sevilla Ruiz
pfc_05@dif.um.es
Facultad de Informática
Universidad de Murcia

Director
Jesús J. García Molina
jmolina@fcu.um.es
Facultad de Informática
Universidad de Murcia

17 de septiembre de 1998

*Este proyecto tiene una asignación de **diez créditos y medio**.

A mi padre.
Su recuerdo es siempre un continuo apoyo.

Índice General

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Los años noventa, la era de la conectividad, han traido consigo que se cambie la forma en la que se desarrollan aplicaciones. Las empresas encuentran más apropiado el uso de aplicaciones distribuidas en lugar de las aplicaciones basadas en *mainframes* o las monoprocesador que hasta el momento venían utilizando. Y esto es así, entre otras cosas, porque las aplicaciones distribuidas permiten que el Sistema de Información se adapte completamente a la estructura de la empresa. Si existen varias nodos en donde se distribuye el trabajo de la empresa, ¿por qué no hacer que el Sistema de Información de la empresa se distribuya de igual manera? Esto permite, entre otras muchas cosas que se verán a lo largo de este proyecto,

- que cada localidad (o elemento estructural de la empresa, por ejemplo, una sucursal de un banco) tenga asignado su parte de funcionalidad del Sistema de Información que, de una forma cooperativa, interactuará con las demás partes para conseguir la funcionalidad general,
- una mejora en la eficiencia del sistema, ya que cada localidad posee los datos que utiliza más a menudo,
- que el resto del sistema siga funcionando aun cuando una localidad deje de funcionar o falle,
- etc.

Pero no todo son ventajas. Las aplicaciones distribuidas son más difíciles de comprender, implementar, depurar y mantener. Por ello se necesitan herramientas potentes que ayuden a los programadores y diseñadores a obtener aplicaciones distribuidas que exploten todo lo posible las ventajas y oculten la dificultad intrínseca de este tipo de aplicaciones.

A la hora de desarrollar e implantar cualquier Sistema de Información (y, por ende, cualquier aplicación distribuida) se necesitan una serie de herramientas y soportes, tanto desde el punto de vista físico como el lógico:

Hardware. Soporte físico, como ordenadores, líneas de conexión y protocolos de bajo nivel: físico, enlace, etc.

Sistemas Operativos de Red. Proveen de un entorno de operación que abstrae a los programas de los detalles de la máquina específica, o de otros programas que se estén ejecutando en la máquina. Ofrecen, además servicios de conectividad que permiten a

los programas, entre otras cosas, acceder a recursos remotos (nodos de procesamiento, impresoras, etc.) de la misma forma que si fueran locales. En definitiva, hacen que los programas se centren única y exclusivamente de la tarea para la que fueron pensados, sin necesidad de tener en cuenta la naturaleza distribuida del sistema.

Protocolos estándar de alto nivel. En entornos donde se requiere que dos o más ordenadores puedan comunicarse para llevar a cabo la tarea, se deben definir una serie de protocolos que estandaricen el modo en el que las aplicaciones existentes en los diferentes ordenadores se comuniquen, independientemente de la aplicación específica.

Modelos conceptuales. Nos permiten establecer un marco lógico que sirve de guía para el desarrollo de las aplicaciones. Por ejemplo, el modelo Cliente/Servidor (sección 2.1 en la página 9) nos permite distribuir la funcionalidad de una aplicación en términos de proveedores y demandantes de servicios.

Tecnologías. Suponen una materialización de un modelo conceptual. Distintas tecnologías pueden aplicar de manera diferente los conceptos de un mismo modelo conceptual.

Lenguajes, entornos de desarrollo y de ejecución. Con los distintos lenguajes y entornos de desarrollo y ejecución se construyen y se implantan definitivamente las distintas aplicaciones que forman el sistema. Éstos darán soporte a distintas tecnologías. Por otro lado, ciertas teconologías, (como RMI y *Servlets*) están restringidas a un lenguaje en concreto, en este caso, Java.

En el contexto de las aplicaciones distribuidas, una elección clara tanto de soporte físico como de protocolos de alto nivel, así como de modelo conceptual (Cliente/Servidor) es Internet (o, a escala de empresa, Intranets). Internet provee un soporte ubicuo para el desarrollo de aplicaciones distribuidas.

Así pues, en este proyecto nos centraremos en el estudio de las distintas tecnologías y lenguajes utilizados para la programación de aplicaciones distribuidas en el entorno Internet/Intranets, tomando además un enfoque que nos permita abordar este estudio de una manera que sea independiente del Sistema Operativo—y, por ello, del hardware subyacente—a través del uso de lenguajes de *script* de alto nivel.

1.1 Objetivos de este proyecto

Este proyecto es un estudio de las diferentes tecnologías que actualmente se están utilizando para el desarrollo de aplicaciones distribuidas en Internet/Intranets, desde las más tradicionales, como los Sockets hasta las de última generación, como los Objetos Distribuidos. Este estudio se hará desde una perspectiva crítica, analizando cada una de las distintas tecnologías en base a unos criterios, como son su eficiencia, el nivel de abstracción que ofrecen al programador y al diseñador de aplicaciones, su escalabilidad, etc. Por lo tanto, veremos si las distintas tecnologías permiten diseñar aplicaciones distribuidas de manera sencilla, reutilizable, extensible, modular. En una palabra, si permiten a las empresas y particulares escribir software rentable, eficiente, fiable, reutilizable y en el plazo de tiempo fijado.

Se persiguen, pues, tres objetivos fundamentales:

- Obtener una visión general del desarrollo de aplicaciones distribuidas en el entorno Internet/Intranets.

- Conocer qué puede y qué no puede proporcionar cada una de las tecnologías, estableciendo una base que sirva de ayuda a quienes se inicien en desarrollos de este tipo de aplicaciones (que, por otro lado, son las que dominan y van a dominar el mercado Internet/Intranets).

- Iniciar una línea de investigación y desarrollo basada en la integración del WEB con los Objetos Distribuidos, lo que en [OH97] es denominado como "ObjectWeb", la nueva ola de desarrollos Cliente/Servidor.

Las distintas tecnologías que se estudiarán son las siguientes: *Sockets*, HTTP/CGI, Java™ *Servlets*, Java™ RMI, Java™ & DCOM y la integración Java™/CORBA.

Estas técnicas cubren la mayor parte de las aplicaciones distribuidas en Internet desarrolladas actualmente y que se desarrollarán en un futuro inmediato. Aún así, es difícil reunir *todas* las variaciones posibles de estas tecnologías, como por ejemplo, distintos productos proporcionados por distintos fabricantes*, (a veces de forma gratuita). Se intentará, en cada sección, introducir algunas alternativas ofrecidas por empresas o particulares que se pueden encontrar por la red. Con esto obtendremos un panorama mucho más amplio y a la vez más *vivo* de cada una de las tecnologías.

El uso de lenguajes de *script*, que tanto auge ha tenido en el mundo Internet, también formará parte importante de este proyecto, estudiando la manera en que este tipo de lenguajes ayuda al desarrollo de aplicaciones distribuidas. Ejemplos y trozos de código que estarán presentes a lo largo de todo el proyecto, ayudarán a mostrar de una forma práctica la teoría que los rodea. Además, la mayoría de las aplicaciones de ejemplo se desarrollan con alguno de estos lenguajes. También se incluye en los apéndices, aparte de todos los listados, una guía personal de iniciación a cada uno de los lenguajes de *script* más utilizados actualmente.

1.2 Qué NO es el objetivo de este proyecto

Aunque el autor de este proyecto ha tenido que conocer a fondo cada una de las tecnologías implicadas, el objetivo de este proyecto no es exponer de manera detallada cada una de ellas a modo de referencia exhaustiva. En cada sección se indica una gran cantidad de bibliografía y sitios WEB que ofrecen información detallada sobre el tema tratado, que ayudará, una vez seleccionada la tecnología adecuada, para lograr llevar a cabo el desarrollo de la aplicación requerida.

La información dada en esta Memoria sirve pues como punto de inicio y de localización de cada tecnología con respecto a las demás. Para una explicación más en profundidad se recomienda la utilización de las referencias.

1.3 Organización del proyecto

En el siguiente capítulo se da una introducción a las aplicaciones distribuidas, estudiando la terminología asociada, los beneficios de las aplicaciones distribuidas sobre las monoprocesador o las basadas en *mainframes*, así como las características de este tipo de aplicaciones, introduciendo de esta manera los problemas y decisiones de diseño que plantean. Como modelo subyacente se introducen los conceptos de Cliente/Servidor. También se estudia Internet como

*como ASP de *Microsoft* o LIVEWIRE de *Netscape*

un gran soporte mundialmente extendido y reconocido para este tipo de aplicaciones. Prestaremos especial atención al uso de los distintos lenguajes de *script* existentes (PERL, TCL/TK, Python), que en el contexto de las aplicaciones distribuidas en Internet está ganando una importancia capital, hasta el punto de que la mayoría de las aplicaciones son desarrolladas con ellos.

En el capítulo 3 se introducen las tecnologías de desarrollo que son independientes de CORBA. En cada sección de este capítulo se estudia una tecnología distinta. Para cada una de ellas, se incluye:

Introducción. Breve introducción a la tecnología en cuestión.

Elementos a manejar. Introduce los elementos a los que el programador se debe enfrentar a la hora de desarrollar una aplicación basada en esa tecnología, como por ejemplo, servidores, sistemas gestores de base de datos (SGBD), interfaces, etc. Sería el equivalente a lo que los anglosajones denominan "*deployment*". Esto dará una idea de los elementos necesarios para *hacer que funcione* un sistema utilizando esta tecnología.

Ejemplo de aplicación. Un ejemplo mostrará el uso en la práctica de cada una de las tecnologías. Será un punto de inicio para introducir las ventajas e inconvenientes.

Ventajas e inconvenientes. Aquí se estudiará la adecuación de esta tecnología comparándola con las demás y teniendo en cuenta los criterios introducidos en el capítulo 2 sobre las características deseables de una tecnología de desarrollo de aplicaciones distribuidas.

Las distintas tecnologías tratadas son las siguientes:

Sockets. Los *Sockets* son la herramienta de más bajo nivel que permite la comunicación entre aplicaciones—y, hasta hace poco, casi la única posible, junto con RPC (*Remote Procedure Calls*) de Sun Microsystems. También es la tecnología más antígua. Los Sockets se tratarán en la sección 3.1. Para mostrar el uso de los Sockets, se desarrollará una aplicación de *charla* (*chat*), con sus dos partes—cliente y servidor—implementadas en distintos lenguajes y/o Sistemas Operativos. Esto mostrará cómo utilizando los Sockets se puede llegar a conseguir, de una manera rudimentaria, la tan deseada "interoperatividad" e independencia del Sistema Operativo, así como la posibilidad de implementar cada parte de la aplicación en la plataforma/lenguaje idóneo.

HTTP/CGI. Con el advenimiento a principios de los noventa de los *browsers* gráficos de páginas WEB (por ejemplo, Mosaic) fue cuando Internet tuvo su mayor auge. Todo el mundo utilizaba estos programas gráficos porque eran más cómodos que los existentes hasta el momento... y también el protocolo asociado: HTTP. Cuando se vió la necesidad de crear un soporte Cliente/Servidor que permitiera desarrollar aplicaciones distribuidas, nació CGI sin un estudio realmente concienzudo. Estos temas se tratarán en la sección 3.2. Para ello, se implementará una aplicación de "listas de correo" o "bulletin board" a través del correo electrónico y el WEB. Esto mostrará varias cosas interesantes: se tendrá que configurar un servidor WEB, un servidor de correo, se tendrá que utilizar el protocolo estándar SMTP [Pos82], se utilizará PERL con el interfaz estándar DBI/DBD y el *driver* para Win32 ODBC accediendo a una base de datos Access a través de SQL.

Java™ *Servlets.* Java ha traido muchos cambios a Internet. Desde su introducción ha revolucionado la manera en la que se diseñan aplicaciones distribuidas en Internet. En este proyecto se tratará Java de manera intensiva, como lo demuestran éste y los siguientes puntos. En particular, los *Servlets* nacieron para cubrir el hueco que en un principio dejó Java en la parte del Servidor, siendo una mejora de los CGIs en ciertos aspectos. La sección 3.3 trata sobre los *Servlets*. La aplicación que se desarrollará aquí será una modificación de la anterior en la que ciertas partes quedan como están y, en otras, se sustituirán CGIs por *Servlets*. Una vez más, este ejemplo mostrará cómo varios lenguajes y tecnologías interactúan de manera que se pueda elegir la adecuada para cada parte de la aplicación. Como servidor WEB se utilizará Java Web Server, mostrando cómo configurar los Servlets y cómo hacerlos funcionar.

Java™ RMI. Con el objetivo de proporcionar una alternativa a los *Sockets*, los creadores de Java introdujeron RMI. La tecnología de Objetos Distribuidos asoma tras las ideas de RMI. No obstante, su uso está limitado a Java y sigue siendo un mecanismo algo primitivo. Todas las características de esta tecnología serán estudiadas en la sección 3.4. En ella, se modificará la aplicación de listas de correo anterior implementando ciertas partes utilizando RMI. También se utilizará el interfaz que Java ofrece para las bases de datos: JDBC para acceder, a través del puente JDBC-ODBC, a las bases de datos creadas utilizando ODBC. Esta sección dará una buena perspectiva de la tecnología que rodea a Java.

Java™ & DCOM. DCOM es la alternativa a CORBA de *Microsoft*. Ambas son tecnologías de Objetos Distribuidos con amplio soporte en toda la comunidad Internet. Un inconveniente de DCOM es que, por el momento, está ligado a entornos Windows. Sus ventajas e inconvenientes se comentarán en la sección 3.5. Se mostrará una sencilla aplicación "ping" que nos servirá para medir el tiempo medio de una invocación utilizando DCOM. Esta pequeña aplicación también servirá para mostrar lo que actualmente es el mayor problema de DCOM (entre otros): su abrumadora configuración.

En el capítulo 4, se dscriben los Objetos Distribuidos resultado de la integración de Java y CORBA. Aquí se presentan las ventajas que esta integración tiene sobre todas las tecnologías competidoras. Primero se introducen los conceptos fundamentales de CORBA, Java y de la integración de ambos. Después se pasa a ver cómo el IDL de CORBA se traduce en construcciones Java. A continuación se presenta una pequeña aplicación Cliente/Servidor clásica utilizando VisiBroker for Java de Visigenic. Esta aplicación mostrará cómo funciona el "ObjectWeb", la integración de Java/CORBA con el WEB. A continuación se ofrecen nociones de conceptos de programación CORBA avanzada, como son las invocaciones dinámicas, los *callbacks*, la interoperatividad entre ORBs de distintos fabricantes y el uso de servicios avanzados. Por último, analizamos un ORB escrito en uno de los lenguajes de *script* más completos: Python.

En el capítulo 5 se ofrecen las conclusiones que este estudio ha generado, realizando una comparativa *a posteriori* de todas las tecnologías utilizadas.

En el apéndice A se muestran los listados de todas las aplicaciones que se han desarrollado, las páginas WEB resultantes, las bases de datos, etc.

En el apéndice B se hace una pequeña introducción personal a los distintos lenguajes de *script*: PERL, TCL/TK y Python.

1.4 Lenguajes, Herramientas y Tecnologías

La realización de este proyecto ha requerido el uso de muchas herramientas, lenguajes de programación y Sistemas Operativos. Esta sección es una descripción del proceso de desarrollo que este proyecto ha tenido, incluyendo la instalación de herramientas, entornos de desarrollo, lenguajes de programación, APIs, interfaces con Bases de Datos, etc.

En primer lugar, se estudiaron los distintos lenguajes de *script* (**Perl**, versión 5; **Tcl/Tk**, versión 8; **Python**, versión 1.5.1; **Java, JDK** versión 1.1.6) para obtener nociones de programación (más avanzadas en algunos casos, como en Java y Perl), centrándonos posteriormente en el interfaz que éstos ofrecen para el uso de **Sockets**. Esto requirió la instalación y el estudio de la documentación de entornos de ejecución (del inglés *run-time environments*) de cada uno de estos lenguajes para cada una de las plataformas utilizadas, **Linux** y **Windows 95/NT**.

Para la parte de programación basada en **HTTP/CGI**, se trabajó con tres servidores WEB: *Netscape Enterprise Server 3.1*, *Java Web Server 1.0.3*, y *Apache* (1.3b3 para Windows 95/NT, 1.2.4 para Linux). Estos servidores fueron configurados para la ejecución de los programas CGI que se ofrecen como ejemplo. Para la realización de estos programas, también se tuvo que estudiar el interfaz estándar de acceso a bases de datos para **Perl** (**DBI/DBD**), el estándar **ODBC**, y el interfaz entre ambos (**DBI::W32ODBC**). También se estudió un interfaz de acceso a bases de datos **ODBC** desde **Tcl** (**TCLODBC**), que permitió reescribir, como ejemplo, algunos programas en Tcl. Por último, para explotar en cierta medida la potencia de procesamiento en el cliente, también se estudió **JavaScript** embebido en páginas WEB que ayudaron en la comprobación de restricciones del interfaz con el usuario.

La programación con **Servlets** requiere un buen conocimiento de la tecnología Java. Para el proyecto, se estudió el conjunto de clases que actúan de soporte para desarrollar *Servlets*, se configuró a los servidores para el uso de *Servlets*, etc.

La sección sobre **RMI** supuso un estudio de las clases específicas de **JDK 1.1** que dan soporte a esta tecnología. Para el acceso Bases de Datos, se utilizó **JDBC**, **ODBC**, y el puente **JDBC-ODBC**, no sin antes estudiar concienzudamente toda la documentación disponible.

Para la sección dedicada a **DCOM**, se requirió el estudio del entorno de desarrollo *Microsoft Developer Studio*, que incluye al *Visual J++ 1.0*. Se estudiaron los distintos *drafts* de *Microsoft* sobre **COM**, **Distributed COM**, **ActiveX**, **OLE 2**, etc. También el uso de herramientas como **MIDL**.

Por último, el capítulo dedicado a **CORBA** requirió de la instalación de tres ORBs, (**Visibroker for Java 3.1**, **OrbixWeb 3.0**, y **Fnorb 0.7.1**, con la correspondiente configuración de todas sus herramientas asociadas. Fue necesario un estudio profundo de los distintos estándares establecidos por el **OMG** (CORBA 2.0, CORBASERVICES, CORBAFACILITIES), prestándo especial atención a las partes específicas de la integración **Java/CORBA**. También, el establecimiento de bancos de prueba que permitieran probar el funcionamiento cooperativo de estos productos en distintas plataformas simultáneamente.

Durante todo el proyecto, y sobre todo para actividades de ordenación y filtrado de *logs* de servidores y estudio de ficheros de texto, se utilizó ampliamente herramientas de administración, tanto de Linux como de Windows 95/NT, como son el shell de Unix (**bash**), **AWK** (por sus autores, Aho, Kernighan y Weinberger), **Perl** (como lenguaje para administración), etc., todos ellos también en sus versiones para Win32 (**CygWin32**).

1.5 La Documentación

La documentación de este proyecto se ha escrito en el sistema de composición de textos TEX de Donald E. Knuth con la ayuda de los macros LATEX 2_ε de Leslie Lamport, utilizando la distribución teTEX versión 0.4 bajo Red Hat Linux 5.0. Las referencias bibliográficas se han mantenido utilizando el programa BibTEX de Oren Patashnik. Para mantener el Índice de Materias se ha utilizado el programa makeindex escrito por Pehong Chen de la misma distribución. Para la adecuación al español del entorno se ha utilizado el paquete Babel cuya versión española es de Johannes L. Braams, junto con los patrones de separación de palabras (*hyphenation*) de Ramón García Fernández. Para algunas figuras sencillas se ha utilizado el programa pic de Brian W. Kernighan en el modo de compatibilidad con TEX. Para la edición de los documentos se ha utilizado emacs versión 20.2.1, con el modo LATEX y el submodo RefTEX, que permite mantener de una manera sencilla todas las referencias introducidas.

Capítulo 2

Aplicaciones Cliente/Servidor Distribuidas

El desarrollo y abaratamiento del hardware, la ampliación de las empresas y la demanda de aplicaciones cada vez más complejas y adaptadas a la estructura de la empresa llevó a que se tuvieran que idear nuevas formas de organizar los SI de las empresas. A medida que el hardware se fue desarollando, la demanda de aplicaciones de gestión automatizada de información fue creciendo. Cuando se necesitaron sistemas de información (SI) que se fueran ajustando a las necesidades de las empresas, la única solución que podían aportar los primeros sistemas, debido en gran parte al elevado coste del hardware, era una configuración en la que un único equipo o *mainframe*—relativamente grande y adaptado a las necesiades de la empresa—gestionaba **todo** el sistema de información. Los distintos puntos en los que se requería el acceso a esa información eran conectados al gran *mainframe* a través de líneas de comunicación utilizando terminales—también llamadas "terminales tontas" o *green screens*—cuyo única funcionalidad era la de mostrar caracteres en la pantalla y enviar la información del usuario en forma de pulsaciones del teclado. Estos terminales eran mucho más baratos que el gran *mainframe*, y, por tanto, una empresa podía distribuir un número relativamente grande de éstos en distintos puntos estratégicos a un coste no muy elevado.

Dos causas llevaron a que esta organización fuera evolucionando: primero, el hardware se hizo cada vez más barato, por lo que en los puntos de acceso de información se podían colocar, a un menor coste, ordenadores con una cada vez más grande capacidad de procesamiento, que quedaba ampliamente desaprovechada al utilizarlos éstos como terminales; sin embargo, la causa más importante era la falta de flexibilidad y escalabilidad de la solución basada en *mainframe*.

En primer lugar, todo el código del sistema junto con sus datos residía en el ordenador principal de la empresa. Esto hacía que, para empresas medianamente grandes, el sistema de información fuera un largo y, a la vez, en muchos casos, incomprensible programa al que los programadores se tenían que enfrentar a la hora de realizar alguna modificación, actualización, etc. En segundo lugar, el sistema quedaba muy poco escalable, y esto era por varias razones: todos los terminales se conectaban a un mismo ordenador, quedando éste limitado *incluso* por el número de interfaces físicos para conexión de terminales de que dispusiera; todos los terminales requerían del servicio del *mainframe* de forma más o menos simultánea, lo que hacía que, al ir añadiendo más terminales, el servicio de las peticiones de cada una de ellas se hacía más lento; etc.

Ampliaciones tan básicas como la inclusión de un segundo *mainframe* por cuestiones de ampliación de la capacidad del sistema o por la apertura de una segunda sucursal, llevaron a que surgieran cuestiones bastante importantes y que no estaban resueltas hasta el momento, como por ejemplo ¿cómo reorganizar la aplicación monolítica existente si en vez de un *mainframe* se tienen dos? ¿Cómo se redistribuyen los datos de estas aplicaciones? ¿Cómo se consigue que ambos ordenadores trabajen juntos y se distribuyan el trabajo para aprovechar realmente ambas capacidades de procesamiento?

De forma paralela, en la parte de los usuarios, equipos cada vez más potentes podían ser instalados, lo que llevaba a la pregunta razonable: ¿cómo se puede conseguir que la capacidad de procesamiento de los ordenadores de los usuarios sea aportada a la del sistema global? En definitiva, ¿cómo **distribuir** la funcionalidad, los datos y los recursos del Sistema de Información de la empresa de una forma que sea escalable y flexible?

Esta pregunta se ha refinado sobre todo en los últimos años—donde la capacidad de procesamiento de los ordenadores de sobremesa dotados de conexiones a redes y de Sistemas Operativos de Red ha sobrepasado la de aquellos *mainframes*—y ha traido otras como: ¿Cómo conseguir sistemas abiertos (en el sentido de que permitan la inclusión de herramientas y subsistemas de terceros de forma sencilla) que permitan integrar los sistemas antíguos o *legacy* de la empresa (posiblemente *mainframes*)? ¿Cómo aprovechar los recursos proporcionados por los ordenadores y los Sistemas Operativos de Red, incluso en sistemas heterogéneos con distintas plataformas hardware y diferentes Sistemas Operativos? ¿Cómo conseguir que la aplicación distribuya la funcionalidad entre todos los equipos de manera que maximice la productividad, capacidad de procesamiento y utilización de recursos y que a la vez permita un diseño modular, reutilizable, basado en objetos, portable, independiente de la plataforma, etc.? ¿Cómo integrar el SI de la empresa con Internet?, etc.

La respuesta a estas preguntas vino a través de la inclusión de un modelo conceptual que separaba los roles de "cliente" y "servidor" para la consecución de una funcionalidad total. Este es el modelo **Cliente/Servidor**, que veremos en la sección siguiente. En la sección 2.2 veremos cómo la tendencia actual adopta a Internet y a las Intranets como soporte para el desarrollo de aplicaciones distribuidas. Por último, en la sección 2.3 estableceremos las bases que nos permitirán evaluar posteriormente las distintas tecnologías que introduciremos en los capítulos 3 y 4. En ellos, veremos cómo las distintas tecnologías se acaptan a los criterios que introduciremos en este capítulo.

2.1 Cliente/Servidor

El modelo Cliente/Servidor ([OHE96], [BJM94], [MZ95], [Lew95]) divide la funcionalidad de una aplicación en torno a dos roles muy bien definidos: "cliente" y "servidor". De modo abstracto, el servidor ofrece una serie de servicios que pueden ser utilizados por los clientes para completar la funcionalidad de la aplicación.

Una interacción básica Cliente/Servidor implica a un cliente que inicia una petición de algún servicio a un servidor, posiblemente incluyendo algunos parámetros que modifiquen el comportamiento del servidor. El servidor entonces realiza la función especificada por el cliente, devolviendo los posibles resultados que el servicio genera.

Esta abstracción permite desarrollar la aplicación en torno a las abstracciones de descomposición modular que proporcionan los servicios.

En la práctica, los clientes y servidores se implementan como procesos que se están ejecu-

tando en máquinas conectadas a una red. La infraestructura que se encarga de conectar de alguna manera los procesos cliente y servidor es llamada *Middleware* (podríamos decir que es la parte "/" de Cliente/Servidor). El término *Middleware* engloba a todos los elementos que hacen posible esta comunicación, desde las líneas físicas de comunicación hasta los protocolos de alto nivel. Como veremos después en la sección 2.2, el soporte estándar de desarrollo de aplicaciones distribuidas es Internet (e Intranet a nivel de empresas), por lo que la infraestructura *Middleware* está cubierta hasta el nivel de transporte de la arquitectura OSI, e incluso en algunos casos hasta el de aplicación. Todas las tecnologías que introducimos en los siguientes capítulos son protocolos que permiten la comunicación de clientes con servidores (esto es, *Middleware*).

2.1.1 Distribución de la presentación, la funcionalidad y los datos

Para poder distribuir una aplicación entre un conjunto de procesadores, es necesario establecer una distinción entre sus distintas "unidades funcionales". Típicamente, una aplicación consta de:

- **Una presentación**, ya sea en base a un GUI o mediante una terminal de caracteres, que implementa la interacción con el usuario y realiza ciertas comprobaciones sencillas (validez de fechas, etc.)
- **Una lógica de negocio o funcionalidad**, que implementa la funcionalidad de la aplicación. Esta lógica de negocio se encarga de realizar todos los procesos que son requeridos por el Sistema de Información.
- **Una lógica de datos**, que establece cómo los datos de la aplicación son estructurados, ya sea a través de bases de datos SQL con tablas, o a través de ficheros de disco.

Cualquiera de estas partes de la aplicación puede ser distribuida. En un sistema con presentación distribuida, cada cliente dispone de su lógica de presentación, y en el servidor se guarda tanto la lógica del negocio como la lógica de datos y los datos mismos. Esta configuración es conocida como una configuración con clientes "livianos" y servidores "gordos" (*thin clients* y *fat servers*). El interfaz entre la lógica de presentación y la de negocio asegura que aunque esta última cambie, los clientes podrán seguir accediendo sin cambio alguno.

Si se elige la distribución de los datos, los clientes poseen ahora, además de la lógica de presentación, la del negocio y la de datos (consultas SQL, etc.), Los datos son distribuidos sobre servidores de manera que se aprovechen al máximo los recursos y características de cada uno. Los clientes se convierten entonces en *fat clients*, que acceden a servidores que se convierten sólo en repositorios de datos. Esta distribución es algo menos flexible, ya que un cambio en la aplicación implica un cambio en los clientes.

La última opción es la más flexible. La distribución de la funcionalidad entre los distintos clientes permite que se aproveche la capacidad de procesamiento de los clientes para participar de forma cooperativa en la consecución de la funcionalidad total de la aplicación. Cada parte de la funcionalidad se puede distribuir en el procesador específico para recibirla, además de poder adjuntarle la lógica de datos y los datos que necesite para llevarla a cabo.

2.1.2 Aplicaciones de dos y tres niveles

Tomando como base la manera en que la funcionalidad se divide entre cliente y servidor, las aplicaciones Cliente/Servidor se dividen en dos grupos: aplicaciones de dos niveles y aplicaciones de tres o n niveles.

Las aplicaciones de dos niveles fueron el primer paso en el desarrollo de aplicaciones Cliente/Servidor. Típicamente, la lógica de presentación, la de negocio y la de datos quedaban en la parte cliente, dejando a los servidores encargados sólo de guardar los datos, es decir, como servidores de Bases de Datos. Esta organización es sin duda un paso adelante con respecto a las soluciones basadas en *mainframes*, ya que permite una cierta escalabilidad y que varios clientes se puedan beneficiar de los datos residentes en los servidores. Sin embargo, no está libre de inconvenientes. Aunque es verdad que a lo largo del cliclo de vida de una aplicación los datos son más estables que los procedimientos, con esta configuración, un cambio en las bases de datos requería la programación de una nueva aplicación cliente y la distribución a todos los usuarios. Esta configuración se muestra en la figura 2.1.

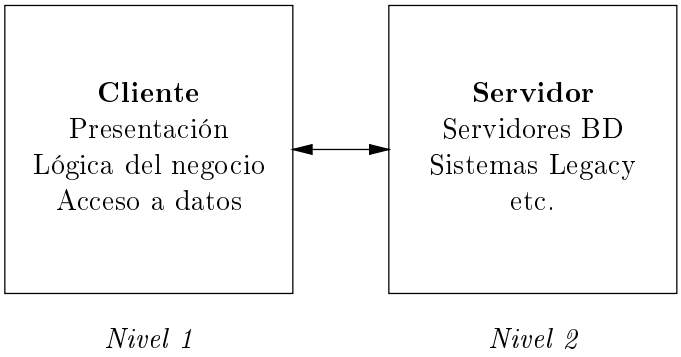

Figura 2.1: Arquitectura de dos niveles Cliente/Servidor

Además, esta solución también presenta una baja escalabilidad, ya que a medida que el número de clientes aumenta, el servidor de bases de datos se ve cargado de forma proporcional al número de clientes. Las aplicaciones son específicas para realizar cierta función y sólo comparten los datos. No se pueden reutilizar partes de aplicaciones ya creadas y se queda ligado al producto utilizado como servidor de base de datos.

Un refinamiento de la arquitectura anterior es la arquitectura de tres ó n niveles. Aquí, el cliente se encarga de mantener el interfaz gráfico de usuario, mientras que existen una serie de componentes intermedios en el sistema que se encargan de implementar la lógica de la aplicación. Por último, hay un último nivel que se encarga del la lógica de datos, típicamente SGBDs o aplicaciones *legacy*. En el momento en el que los componentes de este último nivel se conviertan en clientes de otros componentes, se convierte en una estructura nultinivel. De forma esquemática, esta arquitectura se muestra en la figura 2.2 en la página siguiente.

Esta configuración permite que los clientes se construyan en base a unos servicios encapsulados en los procesos que implementan la lógica de la aplicación, y por lo tanto son más inmunes a cambios tanto en la lógica como en los datos. Aún así, la funcionalidad que los clientes implementan es tan sencilla que los cambios son muy superficiales. Varios clientes pueden reutilizar servicios estándar definidos en el nivel intermedio. La aplicación, al estar dividida en partes más pequeñas, hace que el proceso de distribución de funcionalidad en los

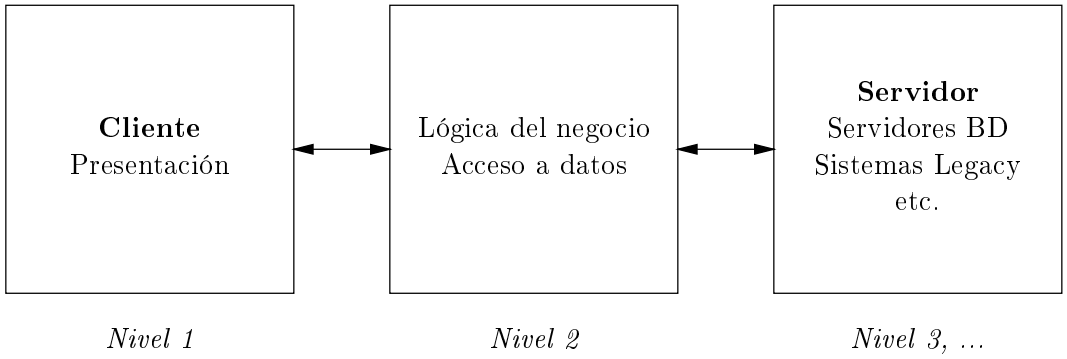

Figura 2.2: Arquitectura de tres niveles Cliente/Servidor

procesadores más adecuados sea muy flexible.

2.1.3 Sistemas Distribuidos

Los sistemas distribuidos son el último paso en la computación Cliente/Servidor ([Lew95], [Ros97]). En vez de diferenciar entre las distintas partes de la aplicación, los sistemas distribuidos ofrecen toda la funcionalidad en forma de "objetos", con un significado muy en la línea del término "Objeto" de la programación Orientada a Objetos. No existen los roles explícitos de "cliente" y "servidor", sino que toda la funcionalidad está ahí para ser utilizada. Los procesos que componen la aplicación y que se están ejecutando en las distintas máquinas de la red son clientes y servidores y cooperan para conseguir la funcionalidad total de la aplicación. Esto da la máxima flexibilidad.

El mundo de los sistemas distribuidos es un mundo de "entidades pares" (*peer-to-peer*), esto es, elementos de procesamiento o "nodos" con distintas disponibilidades de recursos, distinta capacidad de almacenamiento, distintos requerimientos, etc., que cooperan ofreciendo servicios en forma de objetos y requiriendo otros servicios de otros objetos implementados en otros nodos de la red.

En general, un sistema distribuido es un sistema Cliente/Servidor multinivel con un número potencialmente grande de entidades que pueden desempeñar roles de clientes y servidores según sus necesidades. El hecho de ofrecer una serie de servicios en forma de objetos hace que el diseño y desarrollo se haga en base a interfaces bien definidos que facilitan y apoyan la modularidad y reutilización, a la vez que permiten un diseño mucho más flexible.

Los sistemas distribuidos ofrecen, por lo general, un conjunto de servicios añadidos, como el servicio de directorio, que permite localizar servicios por nombre, gestión de transacciones, etc.

2.2 Internet/Intranets: El mayor soporte para programas distribuidos

A nadie escapa el gran auge que está teniendo Internet. Internet es un conjunto realmente grande de equipos (más de 30 millones) conectados entre sí. De ahí que el desarrollo de aplicaciones en este entorno esté tomando una importancia capital. Aplicaciones de reservas de vuelos (de los cuales la más famosa es la de *American Airlines*, http://www.aa.com), venta

por Internet, listas de distribución, etc., se han convertido en habituales, y constituyen la mayor parte de la demanda de desarrollo de aplicaciones.

Desde el punto de vista de la programación de aplicaciones distribuidas, Internet no sólo ofrece una infraestructura física para la comunicación, sino que además ofrece un conjunto de estándares de más alto nivel: ofrece un interfaz de transporte a través de la pila de protocolos TCP/IP; ofrece un servicio de nombrado universal a través de URLs ([BL94]); el protocolo genérico HTTP permite integrar nuevas tecnologías, como los Objetos Distribuidos, a la vez que consigue mantener el interfaz con el usuario basado en el *browser* "universal"...

Una Intranet ([C+96]) no es más que una red de empresa que utiliza los mismos protocolos y estándares que Internet. Lo que implica directamente que está basada en TCP/IP. A partir de esta base, la empresa puede decidir por incorporar más protocolos estándar, como HTTP para utilizar *browsers* para navegar también por la red corporativa. El utilizar los mismos protocolos que Internet premite a la Intranet de la empresa y a todos sus usuarios ver a Internet como una extensión natural de su red corporativa, a la vez que permite controlar de una forma mucho más flexible la información que la empresa ofrece al exterior.

La Intranet integra todos los recursos de la empresa, desde impresoras hasta servidores de Bases de Datos, etc.

2.2.1 El gran papel de los lenguajes de *script*

En el ámbito de Internet y de las aplicaciones distribuidas, los lenguajes de *script* han tenido un auge importantísimo frente a los lenguajes de programación de sistemas. Por ejemplo, la mayoría de los programas CGI, predominantes en Internet, están escritos en Perl. Este auge no es casual ([Ous98]). Los lenguajes de *script* ofrecen la posibilidad de programar a un más alto nivel de abstracción: mientras que el conjunto de tipos de datos en los lenguajes de programación de sistemas está muy orientado a obtener una correspondencia entre los tipos de datos *hardware* (para obtener una mayor velocidad), los lenguajes de *script*, por lo general ofrecen un ambiente en el que las variables pueden ser asociadas a distintas entidades en tiempo de ejecución. Estas entidades no tienen por qué corresponder con tipos de bajo nivel, sino que tienden a ser cadenas de caracteres a las que se le da un significado semántico u otro dependiendo del uso que el programa haga de ellas. Al no estar ligados a características hardware y ser interpretados en el momento de su ejecución*, ofrecen una alta portabilidad entre plataformas, lo que favorece a su vez los sistemas de código móvil (*mobile code systems*), que permiten, entre otras cosas, que trozos de programa o funcionalidad viaje de un nodo a otro para distribuir la carga de trabajo de forma dinámica, que las Bases de Datos almacenen trozos de funcionalidad en forma de *scripts*, etc. Muchos de ellos ofrecen además ambientes de ejecución "seguros" en los que, si no se desea, se inhibe la posibilidad de que el código ejecutable modifique el sistema de ficheros de la máquina en la que se ejecuta. Por último, la mayoría de ellos ofrecen la posibilidad de definición de módulos, clases, sistemas de documentación integrada, metaclases, etc. ([Nic96]). Estas características que facilitan en gran medida la programación de aplicaciones grandes, junto con el aumento de velocidad del hardware y la mejora de las técnicas de compilación dinámica que acercan la velocidad de ejecución de estos lenguajes a la de los de programación de sistemas, hacen que los lenguajes de *script* estén desplazando a los de programación sistemas en el ámbito de las aplicaciones distribuidas ([W3C97]). La figura 2.3 en la página siguiente, extraída de [Ous98], muestra la relación entre

*Aunque, actualmente, los intérpretes de los tres lenguajes de *script* que estudiamos—que son a su vez los más utilizados—realizan primero un proceso de compilación a *bytecodes* como el que hizo famoso a Java

el grado de tipado y el número de instrucciones máquina por instrucción del lenguaje para varios lenguajes de programación tanto de sistemas, como de *script*.

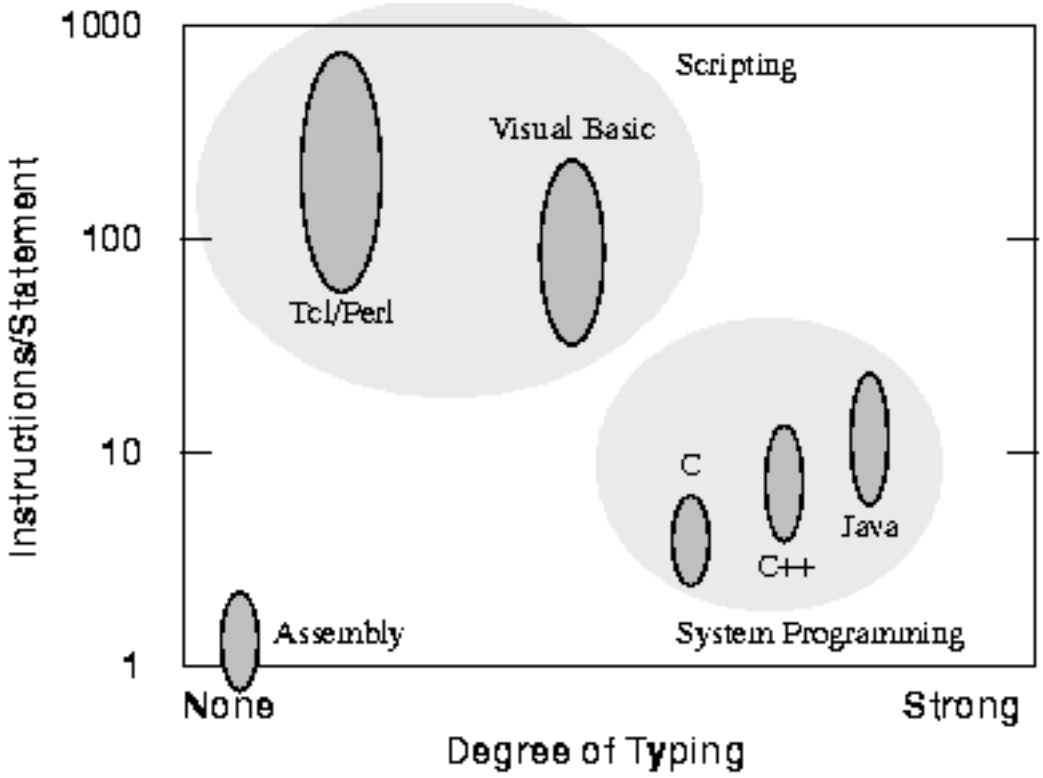

Figura 2.3: Comparación entre lenguajes de *script* y de programación de sistemas (tomada de [Ous98]).

Con un ejemplo podemos obtener una primera aproximación a las diferencias que existen entre la programación en lenguajes de *script* y la programación de sistemas tradicional con C o C++ (incluso Java). Supongamos que queremos escribir el código que realiza la traducción de rutas de directorio que contienen especificaciones relativas a rutas absolutas. Las rutas de directorio relativas se pueden conseguir utilizando los directorios "`.`" y "`..`", subdirectorios especiales de cualquier otro directorio. Por ejemplo, la ruta "`/usr/lib/../bin`" es la misma que "`/usr/./bin`" y la misma que "`/usr/bin`" (suponemos nombrado UNIX). Nuestro código deberá conseguir una ruta de directorio equivalente a la dada y que no use ningún directorio especial.

¿Cómo se escribiría este programa en C?. Es caro que el problema se puede dividir en dos partes o subproblemas: por un lado "arreglar" los dos puntos (`..`), y por otro, el punto (`.`).

- El primer subproblema requiere encontrar, en la cadena original, una subcadena del estilo "`/<caracteres>/../`" (donde `<caracteres>` es un conjunto arbitrario de caracteres) y sustituirla por un único "`/`". Esto, en C, nos obligaría a construir un bucle donde se buscara la secuencia "`/../`", y que guardara el último punto donde anteriormente se encontró un "`/`". A continuación se tendría que sustituir todo ese contenido por un único "`/`", posíblemente utilizando una función que crearía una nueva cadena de

caracteres a partir de la anterior, bien implementada por nosotros, bien obtenida de una librería.

- El segundo subproblema requiere encontrar una subcadena "`/./`" y sustituirla por un único "`/`".

Es obvio que aquí se está haciendo un gran proceso de cadenas basado en bucles cuyos índices pueden irse de los límites, cálculos de diferencias entre índices, etc., que puede llevar muy fácilmente a típicos errores de programación difíciles de eliminar. Además, las funciones sobre las que se basa este procesamiento, si son escritas por nosotros tienen un riesgo potencial de fallar, y si son tomadas de librerías corremos el riesgo de que estas librerías no sean portables entre distintas plataformas, compiladores, etc.

Al otro lado están los lenguajes de *script*. La mayoría de ellos (por no decir todos) ofrecen una librería estándar de proceso de cadenas de caracteres a través de expresiones regulares (entre otras muchas librerías para otros propósitos que después veremos) a menudo muy potentes y que son portables a las mismas plataformas que los intérpretes de los lenguajes (es decir, prácticamente a todas).

¿Cuántas líneas de código se necesitarían para resolver *ambos* subproblemas en **Perl**, por ejemplo? ¿Y cuántos índices y variables temporales? Pensando un poco, el código que resuelve ésto en Perl es el siguiente, suponiendo que la variable "`$dir`" posee el directorio de partida:

```
# -*- Perl -*-
while ($dir =~ s|/[^/]+/\.\./|/|g) {};     # Primer subproblema
while ($dir =~ s|/\./|/|g) {};             # Segundo subproblema
```

¡es decir, dos líneas! La primera línea construye un operador de sustitución para sustituir una subcadena como la que vimos en la exposición del primer subproblema por "`/`" utilizando una expresión regular sencilla al estilo del programa `grep(1)`. El resultado queda de nuevo en la variable "`$dir`". Los bucles se utilizan para prevenir el resultado erróneo que causaría una construcción del tipo "`/usr/lib/../../sbin`" en "`/usr/../sbin`", en la que la sustitución toma en parte a la siguiente aparición.

¿Y en **Tcl**? La respuesta también tiene dos líneas:

```
# -*- Tcl -*-
while {[regsub -all {/[^/]+/\.\./} $dir / dir]} {}
while {[regsub -all /\\./ $dir / dir]} {}
```

A través del comando estándar de Tcl "`regsub`" (sustitución con expresión regular), se especifica que se realicen todas las sustituciones que conformen con el primer parámetro, a la cadena especificada por el segundo, sustituyendo todas las apariciones por el tercer parámetro ("`/`)", y que se escriba el resultado en la variable "`dir`". Las dobles barras inversas (`\\`) de la segunda línea indican que se hacen dos sustituciones en el código: la primera, el intérprete traduce los códigos `\x` (donde "`x`" es cualquier carácter), llevando a la expresión regular a "`\.`". Esta expresión entra en el comando significando lo que queremos: un punto. La barra inversa es necesaria porque el punto es un carácter especial de las expresiones regulares.

Finalmente, ¿cómo se haría en **Python**? Resulta tres líneas más largo, pero la solución queda por el estilo:

```
# -*- Python -*-
import re
n = 1
while n: dir,n = re.subn('/[^/]+/\\.\\./','/',dir)
n = 1
while n: dir,n = re.subn('/\\./','/',dir)
```

La primera línea importa el módulo estándar "`re`", un módulo que implementa expresiones regulares al estilo de Perl. El método `subn` retorna una tupla de dos elementos: la cadena resultado y el número de sustituciones. Se termina el bucle cuando el número de sustituciones, "`n`" llega a cero.

Lo que queda patente en el ejemplo puede ser expresado de una manera más formal utilizando el concepto de *puntos de función* ([Jon97]). Los puntos de función fueron ideados por A. J. Albrecht de IBM como una forma de medir cuán compleja es una aplicación. Hay que hacer especial hincapié en que estas medidas se refieren a la complejidad de la aplicación, independientemente del lenguaje de programación utilizado para implementarla.

En líneas generales, una aplicación mide su complejidad por el número de puntos de función que posee. Para su cálculo, se divide la funcionalidad de la aplicación en torno a cinco unidades bien definidas: el número de entradas a la aplicación, las salidas de la misma, el número de interacciones con el usuario, los ficheros de datos que actualiza y los interfaces con otras aplicacioens. A cada una de estas unidades se le asignó un peso de complejidad relativa calculado empíricamente, entradas por 4, salidas por 5, interacciones con el usuario por 4, actualizaciones de ficheros por 10, e interfaces por 7.

Este resultado nos permite medir la complejidad de los programas. Si medimos además el número de sentencias necesarias para la realización de cada programa, podremos establecer una correspondencia entre el número de sentencias del lenguaje de programación necesarias para implementar un punto de función. Esto nos permitirá comparar la productividad de cada lenguaje de programación, que será mayor a medida que utilicen menos instrucciones para implementar un punto de función. Esta tabla puede encontrarse en [Jon96].

Mientras C necesita 128 instrucciones de media para implementar un punto de función y C++ 53 (al igual que Java), todos los lenguajes de *script* mantienen un número no muy superior a 20. Como comparativa, el código máquina requiere unas 640 instrucciones de media por punto de función. En este sentido, en [Ous98] se presenta una tabla comparativa de programas que se implementaron primero en un lenguaje de programación de sistemas y luego en uno de *script* (figura 2.4 en la página siguiente). Los resultados muestran que el uso de un lenguaje de *script* permitió acelerar de 5 a 10 veces el proceso de desarrollo, incluso tras añadir características adicionales a los programas.

Información adicional se puede encontrar en [Ous98], [Jon96], [Nic96], [LS97], [W3C97], etc.

2.3 ¿Qué estamos buscando?

Desde aquí hasta el final del proyecto analizaremos distintas tecnologías de desarrollo de aplicaciones distribuidas Cliente/Servidor. Éstas serán estudiadas conforme a unos criterios que serán establecidos en esta sección.

Así, las características que consideramos deseables a una tecnología de desarrollo de aplicaciones distribuidas son las siguientes:

Application (Contributor)	Comparison	Code Ratio	Effort Ratio	Comments
Database application (Ken Corey)	C++ version: 2 months Tcl version: 1 day		60	C++ version implemented first; Tcl version had more functionality.
Computer system test and installation (Andy Belsey)	C test application: 272 Klines, 120 months. C FIS application: 90 Klines, 60 months. Tcl/Perl version: 7.7K lines, 8 months	47	22	C version implemented first. Tcl/Perl version replaced both C applications.
Database library (Ken Corey)	C++ version: 2-3 months Tcl version: 1 week		8-12	C++ version implemented first.
Security scanner (Jim Graham)	C version: 3000 lines Tcl version: 300 lines	10		C version implemented first. Tcl version had more functionality.
Display oil well production curves (Dan Schenck)	C version: 3 months Tcl version: 2 weeks		6	Tcl version implemented first.
Query dispatcher (Paul Healy)	C version: 1200 lines, 4-8 weeks Tcl version: 500 lines, 1 week	2.5	4-8	C version implemented first, uncommented. Tcl version had comments, more functionality.
Spreadsheet tool	C version: 1460 lines Tcl version: 380 lines	4		Tcl version implemented first.
Simulator and GUI (Randy Wang)	Java version: 3400 lines, 3-4 weeks. Tcl version: 1600 lines, < 1 week.	2	3-4	Tcl version had 10-20% more functionality, was implemented first.

Figura 2.4: Scripts versus aplicaciones (tomada de [Ous98]).

- Que permita una distribución uniforme de la carga de procesamiento de los distintos nodos implicados en la aplicación. Esto se aplica más en Intranets, donde el software es diseñado teniendo en cuenta los recursos y nodos de que la empresa dispone, aunque también es aplicable en Internet gracias a los sistemas de código móvil (como son los sistemas Java y los lenguajes de *script*),
- Que permita un aprovechamiento máximo de los recursos de cada nodo de procesamiento, o lo que es lo mismo, diseños distribuidos que permitan implantar partes de la aplicación/datos en los procesadores o nodos más adecuados para realizar esa función.
- Independencia del hardware y Sistema Operativo subyacente específico sobre el que las tareas asignadas a un nodo se ejecutan, es decir, que permita la actualización de un

nodo a un mejor hardware o a distinto Sistema Operativo sin que la aplicación deba ser rediseñada ni recodificada, ni, en el mejor de los casos, recompilada.

- Independencia de localización de la funcionalidad. Se busca que una determinada parte de la funcionalidad de la aplicación pueda ser trasladada hacia otro nodo de procesamiento (bien por reestructuración del sistema informático de la empresa, bien por la reestructuración física de los departamentos que la componen) sin que la aplicación tenga que ser modificada (salvo, si acaso, el uso de una herramienta sencilla de reconfiguración automática que indique la nueva localización del recurso). Este punto y el anterior nos aseguran que la aplicación diseñada resistirá a una reestructuración total del sistema informático de la empresa.

- Independencia del lenguaje en el que estén especificadas las distintas partes de funcionalidad que componen la aplicación. Esto nos permite poder utilizar los lenguajes que más se adecúen a cada parte de la funcionalidad: C ó C++ para las aplicaciones críticas en tiempo, Java o lenguajes de *script* como Tcl/Tk para la interfaz con el usuario, etc.

- Integración sencilla con el sistema de información anteriormente existente en la empresa (*legacy systems*), así como de sistemas de terceros, por ejemplo, bases de datos comerciales (*Oracle*, etc.). Construir lo que se conoce con el nombre de *Sistemas Abiertos*.

- Facilitar la instalación de actualizaciones, mejoras, nuevas prestaciones o nuevas versiones de la aplicación. Esto es lo que se conoce como *Gestión de la Configuración*.

- Escalabilidad, o lo que es lo mismo, conseguir que el rendimiento de la aplicación guarde una relación directa con la cantidad de recursos que se le asignen.

- Que permita modularidad, reutilización, extensibilidad, programación en base a interfaces, etc. En definitiva, características presentes en aplicaciones más sencillas que no implican distribución a través de los lenguajes Orientados a Objetos tradicionales.

Una vez establecido qué buscamos de una tecnología de desarrollo de aplicaciones distribuidas, en el siguiente capítulo estudiaremos las tecnologías No-CORBA y veremos hasta qué punto cumplen nuestros deseos. Después se hará lo propio con la tecnología más prometedora del momento: la integración Java/CORBA.

Capítulo 3

Tecnologías No-CORBA

En este capítulo se mostrarán las técnicas de desarrollo de aplicaciones distribuidas que no están basadas en CORBA. Algunas de ellas son viejas conocidas, como los Sockets. Otras son muy recientes como RMI o los *Servlets*. Se dará una breve introducción a cada una y se mostrará una aplicación de ejemplo, que servirá para ver los elementos a manejar, el estilo de desarrollo, las ventajas, inconvenientes y los casos apropiados donde su uso está aconsejado.

En [OH97] también se estudian estas técnicas, pero orientadas sólo a Java y comparándolas con CORBA.

En la sección 3.1 se tratan los Sockets. En la sección 3.2 se estudia el estándar HTTP/CGI. Los *Servlets* se ven en la sección 3.3. En la sección 3.4 se estudia RMI, el soporte de Objetos Distribuidos para Java. DCOM y su integración con Java se verán en la sección 3.5.

3.1 Sockets

En esta sección veremos los Sockets de Berkeley. Los sockets son el primer protocolo entre elementos pares ("*peer-to-peer*")* sobre la pila TCP/IP. Los sockets fueron introducidos en 1981 en la versión 4.2 del Unix de BSD como una manera genérica de comunicación entre procesos (IPC, *interprocess communications*). Actualmente, los sockets son soportados por virtualmente *cualquier* sistema operativo. En Windows, la librería *WinSock* estandariza el uso de TCP/IP bajo Windows. En todos los *nix, forma parte del núcleo. En los demás, se proporciona en forma de librerías. En cuanto a los lenguajes, C proporciona un API que actúa directamente con el núcleo del sistema operativo o con las librerías; Perl provee un API similar al de C; Java proporciona un conjunto de clases para los sockets y otro conjunto para los *streams* asociados a ellos; etc.

Por lo tanto, se puede decir que los sockets son un estándar *de facto* totalmente portable para escribir aplicaciones distribuidas en Internet. Hasta aquí está bien, pero no se debe hacer un juicio sin entrar en detalles. A continuación se verán los sockets con mayor profundidad.

*En la terminología de comunicaciones, se les llama elementos pares a elementos que tienen una función similar. Las dos (o más) entidades que participan en el protocolo son intercambiables. En otras palabras, es un protocolo simétrico. Al contrario, hay protocolos asimétricos, como los basados en Cliente/Servidor, en los que, en cierto momento, una entidad hace de cliente y otra de servidor, con funciones bien definidas y distintas.

3.1.1 Introducción a los Sockets

Un *socket* es un punto de comunicación con una entidad par. En cada extremo, posee un nombre y una dirección de red. En la figura 3.1 se puede ver cómo una comunicación entre entidades pares envuelve a dos sockets, uno en cada parte de la comunicación.

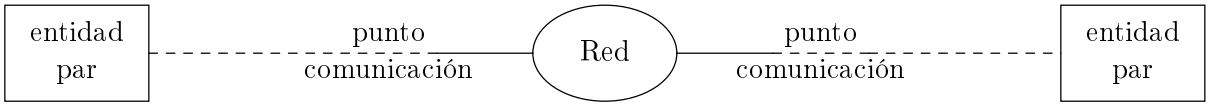

Figura 3.1: Los dos socket en una comunicación entre entidades pares

Desde el punto de vista del programador, un socket abstrae de los detalles de la red. El nivel de transporte, directamente por encima del nivel de red (IP) en la pila de protocolos TCP/IP, ofrece los servicios de sockets a las aplicaciones. Los servicios ofrecidos dan la posibilidad de elegir entre tres tipos de sockets:

Sockets *stream*. Los sockets *stream* están relacionados con el protocolo de transporte más famoso en Internet (la parte TCP de TCP/IP). TCP se encarga de que la comunicación se vea como un *stream* contínuo de bits en el que la aplicación puede introducir y leer datos. TCP también asegura que los datos van a llegar en orden y que se va a mantener la conexión de una manera fiable durante todo el tiempo que dure la comunicación. Es, por tanto, un protocolo orientado a la conexión, lo que significa que tiene una fase de establecimiento en donde se reservan los recursos necesarios para mantener la comunicación, una de comunicación, y otra de liberación de la conexión. Esto es un trabajo difícil, ya que TCP sólo dispone de los servicios de IP, que es no orientado a la conexión[†]. El sistema operativo y los lenguajes de programación ofrecen estos servicios al programador como integrados en los *streams* estándar: escribir a un socket es como escribir a un stream.

Sockets datagrama. Al contrario que los anteriores, este tipo de sockets no ofrecen garantías de ningún tipo. La comunicación a través de los sockets datagrama se estructura en paquetes independientes que se envían al receptor sin ningún tipo de confirmación de que el mensaje llegó sin contratiempos. Los sockets datagrama son una ventana al protocolo de transporte UDP (*User Datagram Protocol*). Como se puede observar, UDP no es orientado a la conexión. Las funciones provistas para manejar este tipo de comunicación incluyen funciones para formar el paquete que se transmitirá, para transmitirlo y recibirlo, y para desempaquetarlo.

Sockets *raw*. Este tipo de sockets no son interpretados por el nivel de transporte, sino que son una ventana directa al nivel de red (IP). Su uso está restringido a cuestiones de administración del nivel de red, como el protocolo ICMP (*Internet Control Message Protocol*), que sirve para informar de problemas en *routers* internos, redirecciones producidas en el paquete transmitido, así como la facilidad *ping*, etc.

La elección de un tipo u otro variará, como siempre, de los requisitos de la aplicación que se vaya a realizar.

[†]Esto no se cubre en este proyecto, pero el lector interesado puede consultar [Ste90].

Independientemente del tipo, cada socket tiene un nombre. La estructuración de la pila de protocolos TCP/IP requiere que a cada nivel, cada entidad tenga un nombre único (figura 3.2).

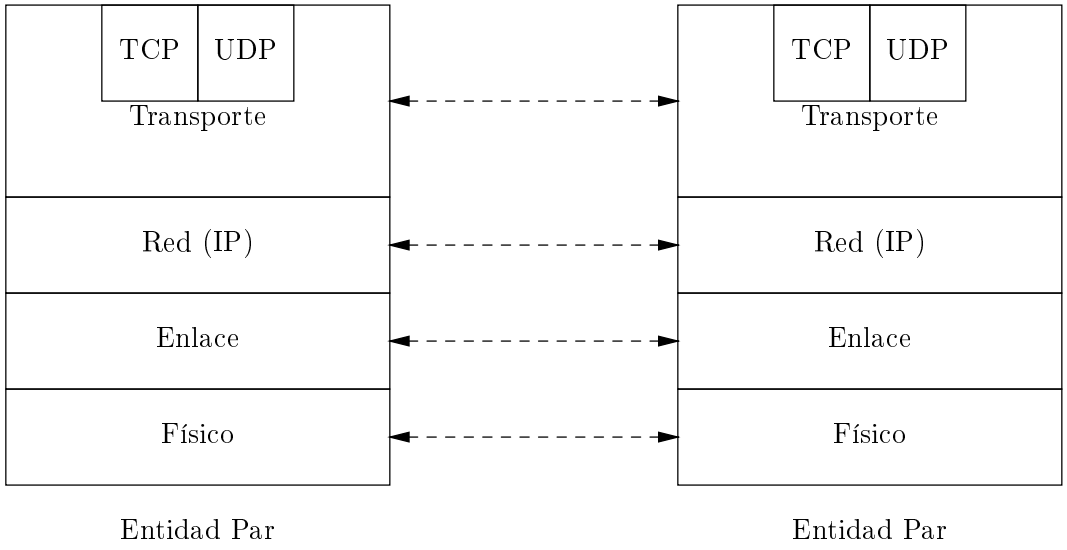

Figura 3.2: Pila de protocolos TCP/IP en dos entidades pares.

Conceptualmente, cada nivel se comunica con el mismo nivel de la entidad par. Las líneas punteadas indican esta comunicación. Los sockets son un servicio ofrecido por el nivel de transporte al nivel de aplicación, por encima de este. Por lo tanto, el nombre de un socket está compuesto de tres partes: la dirección de Red (IP), la dirección de Transporte y el protocolo de transporte utilizado. La dirección de red es una vieja conocida. Es un entero de 32 bits que se representa como cuatro números decimales separados por puntos, como en 155.54.12.132, o, a través del DNS (*Domain Name System*), un nombre que lo identifica, como por ejemplo, o-objeto.dif.um.es. La dirección de transporte es un entero de 16 bits llamado *puerto*.

Existen una serie de puertos cuya numeración se ha estandarizado, estos se llaman *well known ports* (puertos bien conocidos, números del 0 al 1023). Estos puertos se utilizan por las aplicaciones servidor para facilitar que los clientes encuentren los servicios que esperan. Por ejemplo, el servicio HTTP (sección 3.2.1 en la página 33) reside en el puerto TCP 80 (decimal). Dada una dirección IP de un *host* del que queremos acceder al servicio HTTP, basta con conectar un socket TCP al puerto 80 del *host*.

Una manera sencilla de ver el conjunto de puertos bien conocidos a los que da servicio un *host* es el siguiente programa PERL[‡]:

```
setservent(1);
print $nombre,"\t",$puerto,"\n"
    while (($nombre,$dummy,$puerto,$dummy) = getservent);
endservent;
```

[‡]Para una introducción al lenguaje PERL véase la sección B.1 en la página 203.

3.1.1.1 Un escenario de comunicación utilizando sockets

El objetivo de cualquier tecnología de desarrollo de aplicaciones distribuidas es el de, precisamente, abstraer al programador de la naturaleza distribuida de las aplicaciones. Con los sockets se está ante el nivel más bajo de abstracción en la interacción entre localidades en las que la aplicación funciona. Los sockets no constituyen ni siquiera un marco de desarrollo de aplicaciones distribuidas, sino que sólo dan la posibilidad de la conexión: todo lo demás corre a cargo del programador. El estudio de un escenario de comunicación dará la oportunidad de comparar los sockets con otras tecnologías más avanzadas o de más alto nivel que se verán después. Pero no sólo son interesantes por esto, sino porque son la base de todas los demás tecnologías que veremos.

Aunque hasta ahora se han definido los sockets como puntos de comunicación de entidades pares, como ya se vió en la sección 2.1 el modelo de programación que seguiremos es el Cliente/Servidor. Los sockets se adaptan bien a este modelo. Es más, incluyen funciones específicas para ayudar a que una aplicación utilice sus sockets como cliente o como servidor. A continuación se muestra una interacción típica utilizando sockets: se establece un diálogo entre cliente y servidor en donde el primero pide al segundo ciertas informaciones. Los pasos a seguir son los siguientes (figura 3.3 en la página siguiente):

1. **Crear ambos sockets para la comunicación**. Ambos procesos deben crear su socket correspondiente para poder establecer la comunicación. Para esto se utiliza la llamada al sistema `socket`.

2. **El servidor debe stablecer el puerto de servicio**. Esto lo consigue el servidor llamando a la función `bind`. Esto hace que el servidor tenga un nombre único en la red.

3. **Estar alerta de las conexiones de los clientes**. El servidor debe indicar, en el modo de socket *stream* que su socket quedará alerta de las posibles conexiones de los clientes. Esto lo consigue con la función `listen`.

4. **Conectar con el servidor**. El cliente debe entonces, gracias a la función `connect`, conectar con el servidor, especificando la dirección del mismo.

5. **Aceptar la conexión**. El servidor, ha quedado bloqueado por la función `accept` esperando conexiones de los clientes. Para cada conexión, normalmente el servidor crea una nueva tarea, hilo o *thread* para atender a la conexión. El programa original puede volver a ejecutar la función `accept` para esperar nuevas conexiones.

6. **Comenzar la negociación**. Aquí es donde realmente se produce el intercambio de información entre cliente y servidor. Esta parte depende de la aplicación específica. Dependiendo del lenguaje, se utilizarán funciones del tipo `read` y `write`, como en C, o se leerá de los *streams* propios del lenguaje, como en Perl, Tcl o Java.

7. **Cerrar los sockets**. Una vez que se ha terminado la comunicación, ambas partes cierran sus sockets, utilizando la función `close`. Nótese, sin embargo, que el socket original del servidor sigue todavía funcionando esperando nuevas conexiones.

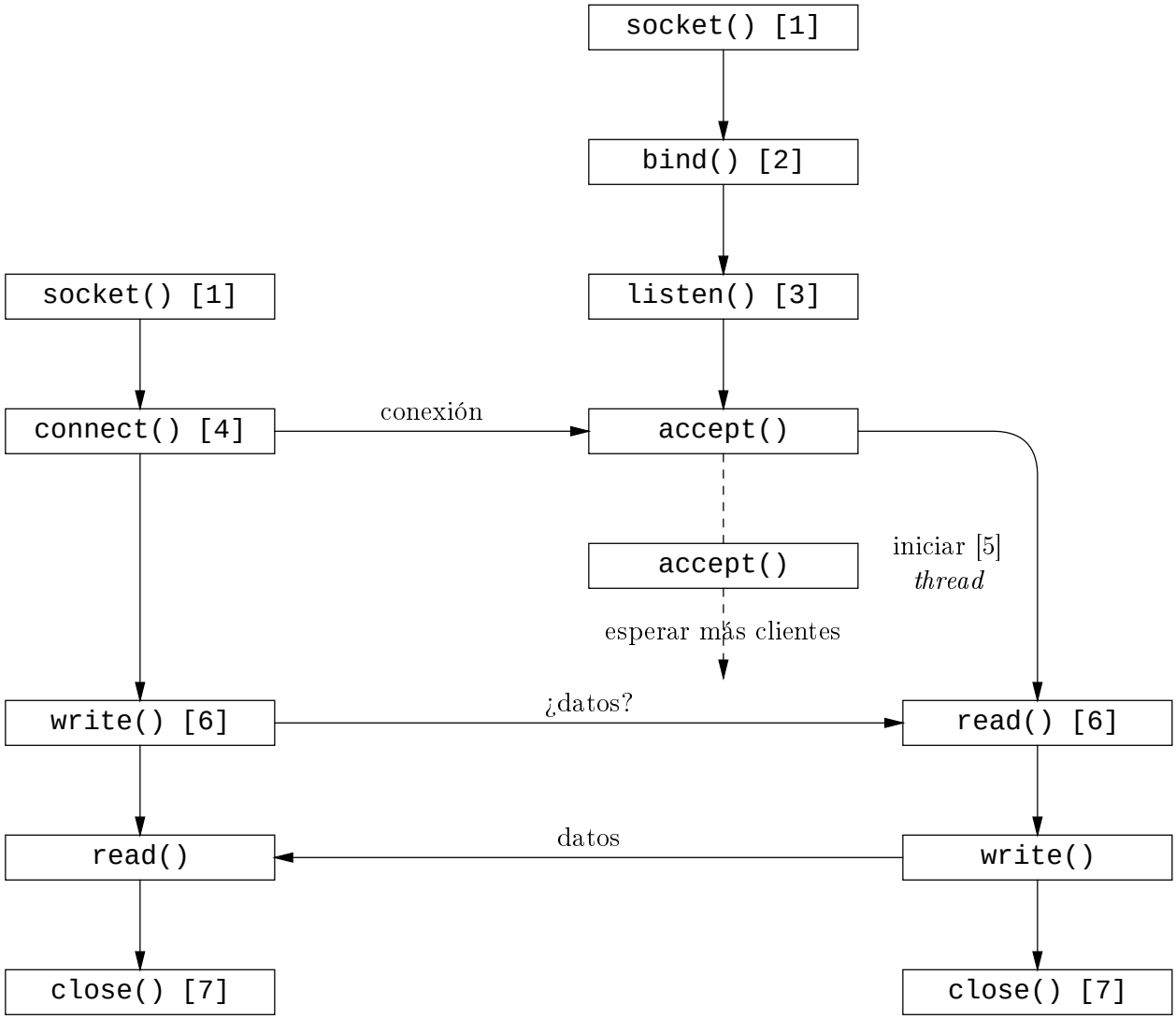

Figura 3.3: Una conversación utilizando sockets entre cliente (a la izquierda) y servidor (a la derecha).

3.1.1.2 Ejemplos de uso de sockets

La sección anterior mostró de forma conceptual un escenario típico de comunicación basada en sockets. En esta sección se verán varios ejemplos de uso práctico de los sockets en varios lengajes de programación.

Perl De todos los lenguajes que se verán, PERL es el lenguaje que trata los sockets de una forma más parecida a como lo hace C, es decir, atacando directamente a funciones de librería o al núcleo del sistema operativo. Por ello, todas las funciones que se vieron en el escenario están presentes en el interfaz que facilita PERL para el uso de sockets.

En primer lugar, hay que incluir las definiciones de las distintas constantes. Éstas se encuentran en el fichero de la librería PERL `Socket.pm`. Para ello, se debe escribir:

```
use Socket;
```

antes de utilizar cualquier función de sockets.

La función `socket()` requiere cuatro parámetros:

```
socket( STREAM, familia, tipo, protocolo )
```

donde `STREAM` es el *stream* que se utilizará para comunicar datos a través del socket; `familia` indica la familia de sockets[§], para este parámetro se utilizará la constante `PF_INET`; `tipo` se refiere a los tipos vistos: *stream* (`SOCK_STREAM`) y datagrama (`SOCK_DGRAM`). El protocolo se refiere a TCP o UDP, los valores para este parámetro se obtienen por la función `getprotobyname('tcp')` ó `'udp'`.

Si la función falla, retorna un valor no definido, y establece el valor de la variable `$!` con el mensaje de error. Un ejemplo de creación de socket puede ser el siguiente:

```
socket(SERVER, PF_INET, SOCK_STREAM, getprotobyname('tcp') ) || die $!;
```

donde se asocia el *stream* llamado `SERVER` con el socket creado.

La función `bind()`, como se vió asocia una dirección (puerto y dirección IP) a un socket. Esta función es utilizada por los programas servidores para asociar un cierto puerto a un socket. La función acepta dos parámetros:

```
bind(STREAM, dirección );
```

donde `STREAM` es el *stream* asociado al socket creado con la función anterior y `dirección` es una dirección de nivel de transporte a la que asociar el socket. Esta dirección debe estar en el formato de bajo nivel que aceptan las librerías. Para ello, existe una función llamada `sockaddr_in()`, que acepta un puerto y una dirección IP. Normalmente un servidor utilizará la construcción `sockaddr_in($port,INADDR_ANY)`, donde `$port` guarda el puerto al que se quiere ligar el socket e `INADDR_ANY` indica la dirección IP de la máquina donde se está ejecutando el servidor. Un ejemplo podría ser el siguiente:

```
bind(SERVER, sockaddr_in( $port, INADDR_ANY ) ) || die $!;
```

La función `listen()` indica que el socket aceptará conexiones de clientes. Las distintas conexiones se quedarán a la espera en una cola. El tamaño de la cola debe ser dado a esta función. Por ejemplo:

```
listen( SERVER, SOMAXCONN ) || die $!;
```

indica la constante `SOMAXCONN` (el número máximo de conexiones posibles) como el tamaño de la cola de conexiones para el socket identificado por el *stream* `SERVER`.

Después de esto, un servidor puede llamar a la función `accept()` para esperar a las conexiones de los clientes. Esta función se bloquea hasta que se produce una conexión. Su signatura es la siguiente:

```
accept( STREAM_NUEVO, STREAM );
```

[§]Hay dos familias de sockets: la familia UNIX y la familia INET (Internet). La primera consigue la comunicación entre procesos a través de ficheros compartidos en la misma máquina. La que interesa en este proyecto es la familia de sockets para comunicación a través de Internet.

Cuando la llamada a la función se desbloquea, en `STREAM_NUEVO` se retorna un *stream* que está asociado con el nuevo cliente que ha iniciado la conexión. `STREAM` es el *stream* asociado al socket del servidor. La función retorna la dirección del socket del cliente. Continuando con el ejemplo:

```
$packedCliAddr = accept( NEW_CLIENT, SERVER );
( $cliPort, $cliIPAddr ) = sockaddr_in( $packedCliAddr );

if (!$pid = fork())
{
    # Hijo sirve al cliente a través del stream NEW_CLIENT

    .
    .
    .

    exit;
}
else
{
    # Padre: debe volver a la primera línea para ejecutar de nuevo el 'accept'
}
```

lo que sirve para que el servidor inicie una nueva tarea con la función `fork()`.

Por su parte, el cliente invocaba a `connect()`. Esta función acepta el *stream* (que debió ser creado también con `socket()`). Si un cliente ha creado el socket `CLIENTE`, y suponemos que quiere comunicarse con el ordenador llamado `un.servidor.es` en el puerto 2005, el código que debería utilizar es el siguiente (las variables `$dummy` se desechan):

```
( $dummy, $dummy, $dummy, @addrs ) = gethostbyname('un.servidor.es');
$packedServiceAddress = sockaddr_in( 2005, $addrs[0] );
connect(CLIENTE, $packedServiceAddress ) || die $!;
```

la función `gethostbyname()` retorna en el array `@addrs` la dirección IP de un *host* dado su nombre.

Una vez establecida la conexión, a través de los *streams*, el cliente y el servidor pueden comunicarse. PERL provee de operaciones sobre los *streams* muy sencillas. la operación `print` se usa para escribir, y, poniendo el *stream* entre ángulos (<>), se lee la siguiente línea, como si fuera un fichero. Continuando con el ejemplo, supongamos que cuando el cliente se ha conectado, espera del servidor una cadena con la fecha y la hora de éste. En la parte del servidor, el programa sería algo como ésto (nótese que en la primera línea se utilizan las comillas inversas para invocar al programa `date` y guardar la salida en `$fecha`):

```
$fecha = `date`;
print NEW_CLIENT $fecha;
close NEW_CLIENT;
```

y la parte del cliente:

```
$fecha = <CLIENTE>;
print "La fecha dada por el servidor es ",$fecha;
close CLIENTE;
```

Una información más detallada se puede encontrar en [HB96, cap. 12] y en [WS92].

TCL/TK En Tcl la cosa cambia, ya que ofrece un interfaz de más alto nivel para los sockets. Esto demuestra lo potentes que los lenguajes de *script* pueden ser para escribir aplicaciones para Internet. Crear un socket de cliente conectando con un servidor en un puerto específico es tan simple como:

```
socket host port
```

donde 'host' puede estar en el formato de nombre o de números separados por puntos. Esta función retorna un identificador de canal (lo que en PERL denominamos *stream*). El mismo cliente de arriba quedaría ahora en Tcl como:

```
set CLIENTE [socket un.servidor.es 2005]
set str [gets $CLIENTE]
puts "La fecha dada por el servidor es $str";
```

donde los argumentos entre corchetes ([]) indican una prioridad en la evaluación, y la función `set` establece el valor de la variable al resultado de la evaluación posterior. Para la parte servidor, la función tiene la siguiente signatura:

```
socket -server comando port
```

donde `comando` es una función que se ejecutará para cada nuevo cliente que se conecte, a la que se le especificará el nuevo canal creado, la dirección de transporte del nuevo cliente (IP + puerto). El mismo servidor anterior se puede escribir en Tcl como:

```
proc acceptClient {ch ip port} {
    set ch2 [ open |date r ]
    set str [ gets $ch2 ]
    puts $ch $str
    close $ch
    close $ch2
}

socket -server acceptClient 2005
```

En `ch2` se crea lo que en la comunidad UNIX se denomina "pipe", que captura la salida del comando `date`, se lee en `str` y se escribe en el canal creado para este cliente, `ch`.

Pero lo más interesante es que un servidor puede estar escrito en un lenguaje y el cliente en otro. Se comunican perfectamente... o no tan perfectamente. Pero esto lo veremos en las ventajas e inconvenientes.

Java Java ofrece una serie de clases, todas del paquete `java.net` que nos ayudan a trabajar con los sockets. En primer lugar, Java ofrece una clase `InetAddress` que encapsula una dirección de Internet y provee métodos para localizar la dirección IP de un *host* dado su nombre, obtener la dirección IP de la máquina en donde se está ejecutando el programa, etc. Las clases que manejan sockets son dos: `Socket` y `ServerSocket`. Java también crea automáticamente dos objetos de las clases *streams* del paquete `java.io`, que asocia con el *socket*. Hay varias clases que implementan distintos tipos de *streams*. Se distinguen entre los de entrada y salida, además de los que utilizan almacenamiento temporal (*buffers*). Existen dos clases abstractas, `java.io.InputStream` y `java.io.OutputStream`, que son las superclases de todos los demás *streams*:

FilterInputStream/FilterOutputStream. Que permite encadenar los distintos *streams*.

DataInputStream/DataOutputStream. Que implementan métodos para leer y escribir de forma cómoda los distintos tipos de datos de Java. Por ejemplo, `readChar()`, `writeDouble()`, etc.

BufferedInputStream/BufferedOutputStream. Interpone un *buffer* en la entrada o la salida, respectivamente.

Gracias a las clases `Filter*Stream`, los *streams* se pueden encadenar de tal manera que, para conseguir que un *stream* se convierta en *buffered*, sólo tenemos que crear un *stream* del tipo `Buffered*Stream` utilizando el anterior como parámetro. Por ejemplo, la clase `Socket`, ofrece un par de métodos, `getInputStream()` y `getOutputStream`, que devuelven, respectivamente, los *streams* de entrada y de salida asociados con el socket. Si queremos añadir un *buffer* al *stream* de salida de un socket, tenemos que crear un nuevo *stream* a partir del anterior:

```
import java.net.*;
import java.io.*;

.
.
.

try {
    Socket sock("un.servidor.es",2005);
    BufferedOutputStream output;

    output = new BufferedOutputStream( sock.getOutputStream() );
} catch (Exception e)
{
    System.err.prinln(e);
    e.printStackTrace();
}
```

En la figura 3.4 en la página siguiente se muestra esta configuración.

Veremos más ejemplos de programación de sockets en Java en la sección dedicada al ejemplo, donde se implementa un cliente y un servidor basdados en sockets. No obstante, el

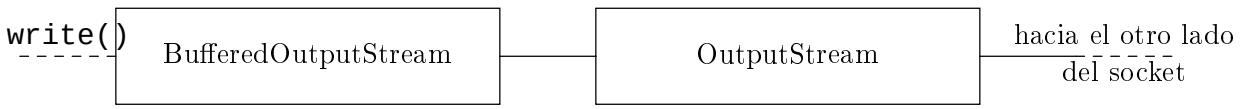

Figura 3.4: Encadenado de *buffers* de salida.

uso de sockets en Java es tratado con mayor detalle en [OH97, cap. 10] y en [Wut96]. La documentación específica de todo el API se puede encontrar en [SUN98].

Python Python ofrece un interfaz para los sockets muy parecido al de C o PERL. Por lo tanto, no se estudiará aquí.

3.1.2 ¿Qué elementos hay que manejar?

No hay nada más que manejar. Una vez que los ordenadores están debidamente conectados a la red e indentificados por su nombre o dirección IP, los sockets pueden ser utilizados sin ningún añadido más. Generalmente las aplicaciones utilizan accesos a bases de datos, implementan interfaces gráficas de usuario (GUIs) en la parte del cliente, hacen chequeos de entrada de datos, etc. Por supuesto que un programador debe enfrentarse a estas áreas, pero éstas son independientes de los sockets.

3.1.3 Un ejemplo de aplicación basada en Sockets

Como se dijo, se verá como ejemplo una aplicación basada en Sockets que implementa un sistema de charla, *chat*. El servidor de esta aplicación está escrito en Java, y su código se puede ver en la sección A.1 en la página 151. Se han implementado, además dos clientes, uno en Java y otro en TCL/TK[¶].

En una máquina se inicia el programa servidor en Java. Los clientes escritos en iTcl, Java o cualquier otro lenguaje o Sistema Operativo se pueden conectar a ella, en el puerto 9000. El trabajo del servidor es simple: debe aceptar nuevos clientes y estar alerta de los mensajes que estos le envían, para reenviárselos a los demás clientes que participan en la charla. La topología queda en estrella como se muestra en la figura 3.5 en la página siguiente. Cada vez que un usuario escribe una línea de texto, la aplicación cliente envía el texto escrito junto con la identificación del usuario por el socket hacia el servidor. Éste, entonces, utiliza el canal de comunicación que tiene con cada cliente para reenviarle a todos ellos esa línea de texto.

En la figura 3.6 en la página siguiente se puede ver una ventana iTcl de la aplicación de charla. En la figura 3.7 en la página 30 se puede ver la misma aplicación como un applet de Java. Todos los clientes comparten el mismo diseño de interfaz. En la parte superior se muestra un cuadro de texto que invita al usuario a poner su nombre para ser identificado por los demás que utilicen la aplicación. El panel central recoge todas las líneas escritas por todos los usuarios en todos los clientes. El cuadro de texto inferior es donde el usuario actual puede escribir sus textos.

Pero lo que más llama la atención de esta aplicación es que se pueden utilizar de forma intercambiable tanto Sistemas Operativos como lenguajes. El servidor y los clientes, aunque estén escritos en diferentes lenguajes y se ejecuten en distintos Sistemas Operativos, son

[¶]En realidad, en [incr Tcl]. Véase sección B.2 en la página 206.

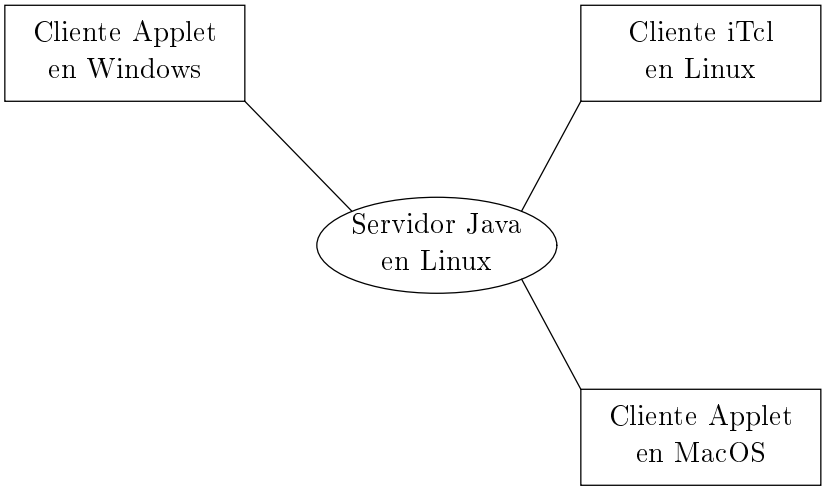

Figura 3.5: Un ejemplo de topología de la aplicación de charla.

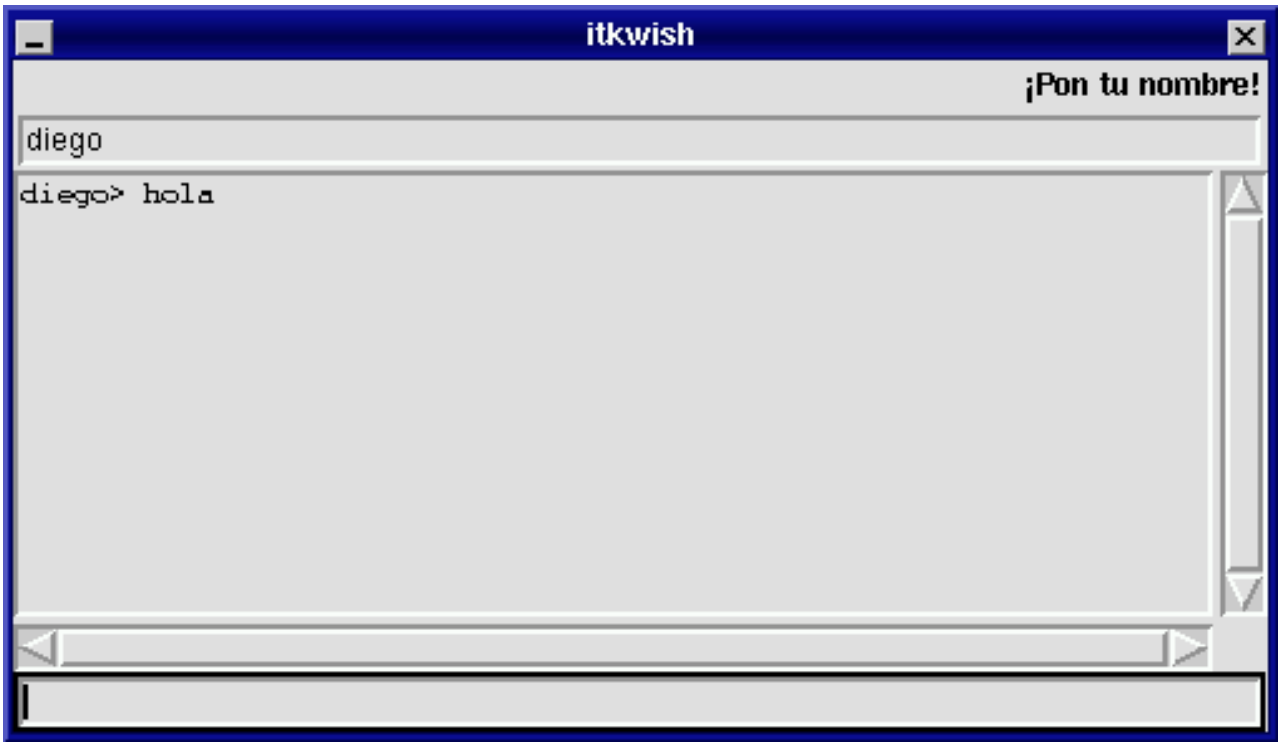

Figura 3.6: Una ventana Tk del *chat*.

capaces de comunicarse sin ningún problema. Esto es, como vimos, la tan deseada interoperatividad.

Si profundizamos un poco más, sin embargo, nos damos cuenta de que esta interoperatividad es sólo aparente. Esta aplicación funciona bien porque sólo transmite cadenas de caracteres entre las aplicaciones. En la mayoría de los sistemas, se ha adoptado el código ASCII o el ISO-5589-1 (ISO-Latin-1), por lo que el texto escrito será *muy parecido* en todos los clientes. ¿Pero qué podríamos hacer si quisiéramos transmitir un entero? No podemos

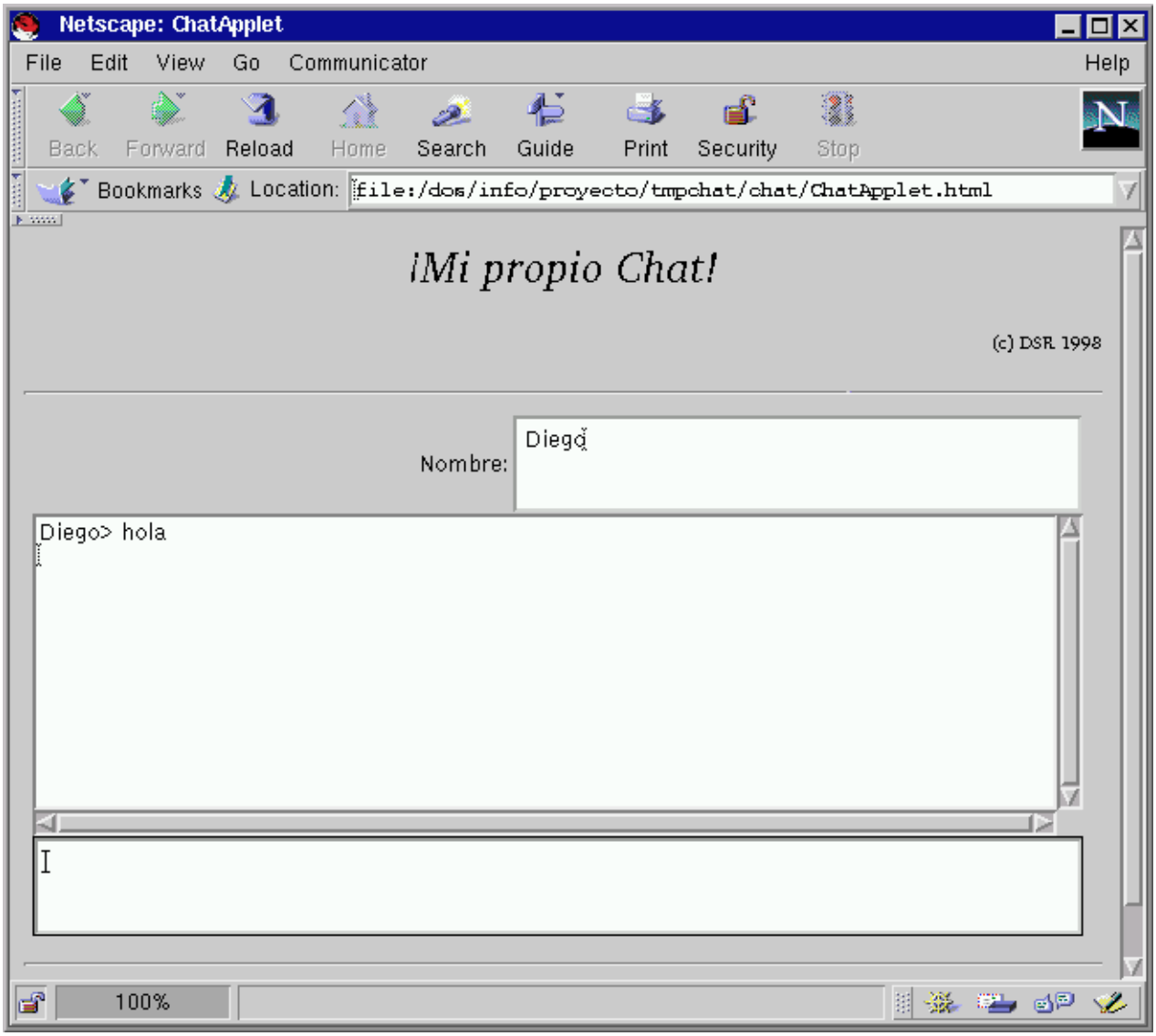

Figura 3.7: El *applet* de charla.

transmitir la representación interna de la máquina, ya que existen máquinas *little endian* y *big endian*, por ejemplo, con lo que tenemos dos opciones:

- transmitir la representación ASCII del entero por la red. Esto nos supondría cinco o seis bytes más uno de terminación, ó
- codificarlo con lo que se denomina *network byte order*, una codificación independiente de la máquina para datos binarios.

Hasta aquí bien, pero ¿y para otros tipos de datos, como por ejemplo, los `double`? ¿Y los tipos compuestos? ¿Y los objetos? La cruda realidad es que el programador debe idearse un sistema de comunicación para cada plataforma hardware+Sistema Operativo con la que vaya a trabajar que estandarice los tipos de datos a transmitir. No hay estándares, a no ser que se

trabaje con un mismo lenguaje y con la misma plataforma en ambos, cliente y servidor. Pero esto no es lo que estamos buscando.

Pero no nos olvidemos de nuestro propósito. Nuestro objetivo es analizar estas tecnologías a la hora de programar aplicaciones distribuidas. Es muy normal que queramos obtener varios servicios de cada servidor. Debemos tener una manera de identificar a los distintos servicios por un nombre. Además, cada servicio requerirá unos parámetros y devolverá unos resultados. Visto así, se puede considerar como un conjunto de funciones, con sus parámetros y sus valores devueltos. Pues sí. Consideremos un conjunto de funciones a las que un servidor basado en sockets tiene que atender. Cada operación tiene un nombre, unos argumentos y unos resultados. Una llamada a una operación en el servidor debería incluir el nombre de la función, y después, cada uno de los argumentos debidamente codificado[‖]. El servidor leería del *stream* el nombre de la operación, y, dependiendo de la misma, debería leer y decodificar cada uno de los parámetros. Entonces debería ejecutar la operación, codificar los resultados y enviárselos al cliente que inició la petición. Por desgracia, este soporte lo debe proporcionar el programador. Si nos imaginamos ahora un sistema con cientos de funciones con varios parámetros cada una, el diseño y mantenimiento puede ser una auténtica pesadilla.

3.1.4 Ventajas e inconvenientes

Con los sockets estamos al nivel más bajo de comunicación. A este nivel es posible implementar todos los servicios necesarios. Pero esto mismo se convierte en un problema ([SV95a] y [SV95b]), ya que es considerable la cantidad de circunstancias de bajo nivel con las que nos tenemos que enfrentar. Para construir aplicaciones distribuidas de forma seria, esperamos un mayor nivel de abstracción, que se nos ofrezcan una serie de servicios estándar adicionales que hagan que la aplicación se centre sólo en su cometido. En definitiva, los puntos estudiados en la sección 2.3.

Se verán a continuación las ventajas e inconvenientes de la programación usando sockets. La introducción y la aplicación de ejemplo dieron una idea de esta tecnología y nos sirvieron para analizar ventajas e inconvenientes. Entre las **ventajas** se pueden citar:

- ✔ **La comunicación a través de sockets es muy rápida**. En [OH97] se realiza una prueba de *ping* utilizando dos Pentium 120 MHz. ejecutando Windows NT 4.0 sobre una red Ethernet a 10 Mbps. y se obtienen los siguientes resultados:

Sockets Locales (entre procesos)	**Sockets Remotos (Ethernet a 10 Mbps.)**
1.8 milisegundos	2.0 milisegundos

Tabla 3.1: Tiempo *ping* para sockets.

- ✔ **Son el estándar *de facto* para programación en Internet**. No olvidemos que la mayoría de protocolos existentes en Internet (como HTTP, SMTP, etc.) utilizan sockets para establecer la comunicación. Virtualmente todos los Sistemas Operativos y lenguajes de programación ofrecen soporte para programar con sockets.

[‖]En jerga de Objetos Distribuidos, a esta codificación se le llama *marshaling/unmarshaling*.

✔ **Su programación es relativamente sencilla para aplicaciones no muy complejas**. Por ejemplo, diseñar un sistema de consultas a Base de Datos en SQL que envíe las consultas del tipo "`select * from ... where ...`" y que el resultado se retorne como una página de sólo texto o HTML es cuestión de minutos, además de ser portable. Pero ¡esto sólo se da en sistemas pequeños!

Durante el recorrido por los sockets hemos visto algunos de sus problemas. Aquí quedan resumidos algunos de sus **inconvenientes**:

✘ **No ofrecen un sistema de meta-información**. Las aplicaciones clientes deben ser configuradas con la dirección IP y puerto de la máquina donde reside el servicio deseado. Dejar a los usuarios este tipo de configuración hace que éstos se tengan que enfrentar a características de bajo nivel. Además, cuando un servicio cambia de localización, todos los clientes deben ser reconfigurados. No existe un servicio de directorio distribuido que mantenga meta-información ni de la localización de servicios ni siquiera de los servicios existentes. Esto es un problema, por ejemplo a la hora de añadir más servicios, que hace que todas las aplicaciones clientes se deban cambiar. Pero no sólo eso, sino que no existe información sobre los servicios que cada servidor ofrece (siguiendo con la sección anterior, los métodos o funciones), es decir, no hay meta-información sobre los interfaces de cada servidor. En el capítulo 4 veremos cómo todos estos servicios, así como un desarrollo de objetos distribuidos basado en interfaces independientes del lenguaje de programación son ofrecidos por CORBA como parte del conjunto de servicios básicos que proporciona.

✘ **No ofrecen una estandarización de tipos**. Los sockets no ofrecen un conjunto de tipos estándar. El programador se tiene que encargar de la codificación y decodificación de los parámetros de los datos transmitidos (*marshaling*). Esto trae una serie de consecuencias indeseables:

- hace que la codificación de la aplicación resulte tediosa, compleja y propensa a errores, al tener que implementar cómo se codificará y decodificará cada tipo de datos y cómo lo harán los tipos compuestos: registros, arrays, etc.,
- hace casi imposible, para sistemas grandes, el mantener un conjunto de servicios que utilicen distintos tipos de datos,
- la depuración se hace extremadamente difícil, al no poder tener ni siquiera un chequeo del número y tipo de parámetros de cada función o servicio.

✘ **Los errores son de muy bajo nivel**. Cuando se trabaja con sockets, los errores que se obtienen son del tipo "*host* desconocido" o "no alcanzable", "se ha perdido la conexión con el *host*", etc. Ningún otro tipo de error, por ejemplo, número de argumentos erróneo, o tipo de algún parámetro equivocado.

✘ **El código se hace poco portable**. Debido a la gran cantidad de código de bajo nivel que hay que escribir, las aplicaciones basadas en sockets, a la vez que se van haciendo más grandes, son cada vez menos portables, hasta el punto de tener que escribir el código de codificación y decodificación de tipos de datos específicamente para cada plataforma utilizada.

Así pues, parece que los sockets no se acercan siquiera a las características que establecíamos como deseables para una tecnología en la sección 2.3. Aparte de su dificultad intrínseca y de su bajo nivel de abstracción, no nos permiten construir aplicaciones que sean realmente reutilizables, ni siquiera en la parte del servidor, más estática, ya que los distintos servicios que éste acepta deben ser identificados y descompuestos sus argumentos en el servidor mismo. Es decir, no se puede separar el manejo de los distintos servicios de la implementación de los mismos, lo que hace que las aplicaciones queden diseñadas *ad hoc*, y no reutilizables.

En la siguiente sección veremos una tecnología que sube un poco el nivel de abstracción con respecto a los sockets. Aún así, podemos adelantar que tampoco será la solución ideal que estamos buscando.

3.2 HTTP & CGI

A un nivel de abstracción por encima de los sockets, nos encontramos con la combinación de HTTP y CGI. Esta tecnología es la que actualmente domina el desarrollo de aplicaciones distribuidas en Internet. Desde la aparición de los *browsers* gráficos, el lenguaje HTML y el protocolo HTTP, los desarrolladores disponen de una plataforma que estandariza el cliente (el *browser*) y permite generar vistosos resultados en páginas HTML con gráficos. Se estableció además un sistema de nombrado y localización de recursos y servicios estándar: URL (*Universal Resource Locator*) [BL94]. Este sistema de nombrado es universal y estándar, lo que significa que no sólo sirve para nombrar ficheros HTML, sino también servicios. Sin embargo, sin un *interfaz* que permita conectar esos nuevos servicios con un servidor WEB, el desarrollo de aplicaciones distribuidas no sería posible en este entorno. Para ello se creó el *interfaz* CGI (*Common Gateway Interface*), que permite conectar a las aplicaciones servidor con el servidor WEB y les permite a las primeras producir páginas HTML—perfectamente interpretadas por los *browsers*—como respuesta a las peticiones de los clientes.

En esta sección entraremos en el mundo de los servidores WEB y de la programación CGI. Comenzaremos con una pequeña introducción tanto a HTTP como a CGI, que nos servirá para, en la siguiente sección, desarrollar una aplicación de listas de correo o *bulletin board* a través del sistema de correo de Internet (SMTP, POP3) y del WEB.

3.2.1 Introducción a HTTP

HTTP (*HyperText Transfer Protocol*) es un protocolo Cliente/Servidor construido sobre los Sockets. Fue introducido en 1990, siendo su primera versión, HTTP/0.9, un protocolo muy simple de transmisiones de datos *raw* o sin formato. HTTP/1.0 vino a mejorar al anterior introduciendo el formato de codificación MIME (*Multipurpose Internet Mail Extensions*) [FB96]. La nueva versión, HTTP/1.1 añade nuevos servicios, la posibilidad de reiniciar comunicaciones en el punto donde se dejaron y la posibilidad de conexiones "Keep-Alive", que después veremos.

HTTP trabaja, como todos los protocolos Cliente/Servidor utilizando un modelo petición/respuesta. Un cliente o *browser* inicia una petición que es enviada al servidor. Éste, por su parte, debe llevar a cabo la acción requerida y enviarle la respuesta que la acción generó. HTTP es un protocolo estándar, orientado a objetos, reconfigurable y sin estado. ¿Y qué significa esto de cara al programador? Bien:

- HTTP es estándar y es utilizado *todos* los días por el 100% de usuarios y servidores (*hosts*) de Internet.
- Cada URL se considera como un objeto al que se le puede pedir la ejecución de un método, adjuntando unos parámetros y, recibiendo en la respuesta unos resultados.
- En el contexto de los servidores WEB, existen un conjunto de métodos que tienen un comportamiento predecible y uniforme. Sin embargo, un objeto en particular puede aceptar un conjunto distinto o aumentado de métodos. HTTP provee "huecos" por donde se puede invocar a estos nuevos métodos.
- Al no mantener el estado, un servidor HTTP no recuerda peticiones ni clientes anteriores, por lo que se deben idear métodos (en ciertas ocasiones truculentos, como los *Cookies* de Netscape Corp.) que permitan mantener la simulación de una especie de "transacción" de una cierta duración.

Esto hace a HTTP un protocolo de interés (salvo el último punto, al que atacaremos en profundidad). Sin embargo, parte de su interés se centra en que es un protocolo tan genérico que se puede aplicar a cualquier ámbito. Aporta poco a la programación sobre sockets. De hecho, la comunicación HTTP se hace a través de sockets. Sin embargo, sí que hay un punto de avance en HTTP: la estandarización de los tipos de datos que los mensajes pueden contener.

3.2.1.1 Codificación de datos en HTTP

En la sección dedicada a los sockets vimos que el principal inconveniente de su programación era su bajo nivel de abstracción, de lo que se derivaba que no existían estándares de codificación de los datos transmitidos. HTTP avanza en este sentido al adoptar una modificación del formato MIME. Esta adopción obliga a que todos los mensajes (tanto las peticiones como las respuestas) que se envían a través de HTTP contengan una identificación que espeficique el *tipo* de la información que contienen. Los tipos MIME se representan con una división en *tipo/subtipo*. Por ejemplo, las páginas HTML tienen un tipo *text/html*. Existe una serie de tipos estándar, y otros nuevos pueden ser utilizados, en cuanto cliente y servidor se pongan de acuerdo.

Esta estandarización y auto-descripción utilizando MIME resuelve el problema de la estandarización de los tipos de datos, pero introduce otros, como por ejemplo, los programas se vuelven más complejos y se introduce una gran sobrecarga (*overhead*)**, su restricción a usar caracteres del código ASCII de 7 bits, etc., como veremos en la sección dedicada a las ventajas e inconvenientes.

3.2.1.2 Peticiones HTTP

HTTP utiliza los sockets para transmitir tanto sus peticiones como sus respuestas. Es un protocolo simple, con una respuesta para cada petición. Ambas están estructuradas en líneas de texto, es decir, cadenas de caracteres (ASCII de 7 bits) separadas por los caracteres de fin de línea ("`\r`", "`\n`", también llamado "`crlf`").

Una petición HTTP consta de una línea de petición, unos *headers* con datos sobre la petición y un cuerpo de la petición.

** o relación $\frac{bytes\ de\ datos}{bytes\ de\ control\ o\ totales}$

- **La línea de petición** contiene tres campos de texto separados por uno o más espacios. El primer campo, especifica el método que se aplicará al objeto del servidor especificado por el segundo campo de esta línea. Los métodos más comunes en el ambiente de los navegadores son, por ejemplo, GET, que pide una copia del recurso, POST, que envía datos al recurso especificado, etc. En la tabla 3.3 se verán los distintos métodos estándar HTTP. El tercer campo especifica la versión del protocolo utilizado por el cliente, por ejemplo, "`HTTP/1.0`".

- **Los *headers*** contienen datos sobre la petición, identificando al cliente, el tipo de petición, etc. Son conjuntos de pares (nombre,valor), separados por dos puntos (`:`). El orden de los encabezados no es esencial. Después veremos los distintos encabezados que pueden usarse. Los headers terminan con una línea en blanco, es decir, que sólo contiene un `crlf`.

- **El cuerpo de la petición (*entity body*)** contiene información que los clientes pasan al servidor en la petición. Con un formato conveniente, se pueden convertir en los parámetros de entrada para el servicio (argumentos de función).

En la tabla 3.2 se puede ver un ejemplo de petición HTTP.

Sintaxis	**Ejemplo**

<método> <recurso (URL)> <versión HTTP><crlf>
<header>: <valor> <crlf>]
⋮
<crlf>
cuerpo]
	`GET /path/fichero.html HTTP/1.0`
	`User-Agent: Mozilla/4.05 [en]`
	`Accept: text/html`

Tabla 3.2: Ejemplo de petición HTTP

En la tabla 3.3 se muestran los métodos HTTP estándar. Se muestra también cuales de ellos son válidos en HTTP/1.0 y HTTP/1.1.

En las tablas 3.4 a 3.7 se muestran los *headers* generales, los específicos de las peticiones, los específicos de las respuestas y los que se refieren al cuerpo del mensaje.

3.2.1.3 Respuestas HTTP

Una respuesta HTTP cuenta con: una línea de estado, un conjunto de *headers* y un cuerpo del mensaje, que contiene la respuesta en sí, codificada utilizando MIME:

- **La línea de estado** retorna la versión HTTP utilizada por el servidor, el estado de la respuesta (como un número seguido de una explicación del estado como una cadena ASCII). Los distintos códigos se dividen en grupos ([FGM+97]):

  - 1xx: Informativos
  - 2xx: Peticiones correctas
  - 3xx: Redirección
  - 4xx: Error del cliente

Método	HTTP/1.0	HTTP/1.1	Descripción
GET	√	√	Recupera el URL especificado
HEAD	√	√	Idéntico a GET, excepto que el servidor no retorna el documento en respuesta; Sólo retorna los headers de respuesta. Los clientes lo utilizan para obtener "meta-datos" del recurso o para saber qué links son válidos.
POST	√	√	Envía datos a la URL especificada.
PUT		√	Guarda estos datos en el URL especificado, reescribiendolo.
PATCH		√	Similar a PUT, excepto que contiene una lista de diferencias entre la versión original del recurso y el contenido deseado.
COPY		√	Copia el contenido del recurso a la(s) direccion(es) especificada(s).
MOVE		√	Mueve el contenido del URL a otra(s) posición(es).
DELETE		√	Borra el recurso identificado por el URL.
LINK		√	Establece una o más relaciones de ligadura (*link*) entre el URL y otros recursos.
UNLINK		√	Elimina la relación existente entre este URL y otro(s).
TRACE		√	Responde, en el cuerpo del mensaje, la petición del cliente.
OPTIONS		√	Pide información acerca de las opciones de comunicación disponibles para el URL especificado. Permite a un cliente obtener las características de un servidor sin recuperar ningún URL.
WRAPPED		√	Permite que las peticiones se agrupen y, posiblemente se encripten para aumentar la seguridad de las mismas.

Tabla 3.3: Métodos HTTP

Header	HTTP/1.0	HTTP/1.1	Descripción
Cache-Control		√	Contiene directivas sobre *caching* de la petición/respuesta.
Connection		√	Indica parámetros sobre la conexión, como por ejemplo, "Keep-Alive", que permite varias peticiones/respuestas utilizando sólo una conexión, y por ello ahorrando recursos.
Date	√	√	Contiene la fecha y la hora en la que el mensaje fue originado.
Keep-Alive		√	Contiene información de diagnóstico.
MIME-Version	√	√	Contiene la versión MIME utilizada para codificar el mensaje.
Pragma	√	√	Contiene directivas de implementación.
Upgrade		√	Lista protocolos de comunicación adicionales que el cliente usa y que le gustaría que el servidor utilizara.

Tabla 3.4: Headers HTTP generales

Header	HTTP/1.0	HTTP/1.1	Descripción
Accept		√	Lista de los tipos/subtipos MIME aceptados.
Accept-Chars		√	Indica qué códigos de caracteres se aceptan.
Accept-Encoding		√	Lista las codificaciones aceptadas, como "compressed" y "zip", etc.
Accept-Language		√	Lista los lenguajes naturales aceptados.
Authorization	√	√	Pasa los esquemas de autentificación y encriptación del usuario.
From	√	√	Contiene el e-mail del usuario.
Host		√	Contiene el nombre del *host* destino.
If-Modified-Since	√	√	Contiene la condición de tiempo para GET.
Proxy-Authorization		√	Permite a los clientes presentar su autorización a un *proxy*.
Refer	√	√	URL del documento desde donde se originó esta petición.
Unless		√	Lista condiciones en el header que deben cumplirse antes de que el método se aplique al objeto.
User-Agent	√	√	Indica el *browser* que utiliza el cliente.

Tabla 3.5: Headers HTTP de petición

Header	HTTP/1.0	HTTP/1.1	Descripción
Location	√	√	Informa de la localización exacta del recurso. Normalmente se utiliza para redirigir a los *browsers* de forma automática hacia otro recurso.
Proxy-Authenticate		√	Retorna el esquema de encriptación/autorización que se está utilizando en esta sesión.
Public		√	Lista todos los métodos no-estándar soportados por un servidor.
Retry-After		√	Indica (en segundos) cuándo reintentar un servicio.
Server	√	√	Identica el software del servidor.
WWW-Authenticate	√	√	Indica qué esquema de encriptación/autorización quiere seguir el servidor.

Tabla 3.6: Headers HTTP de respuesta

Header	HTTP/1.0	HTTP/1.1	Descripción
Allow	√	√	Lista los métodos aceptados por este URL.
Content-Encoding	√	√	Indica la codificación de la respuesta.
Content-Language		√	Indica el lenguaje natural de la respuesta.
Content-Length	√	√	Longitud en bytes del cuerpo.
Content-Type	√	√	Tipo MIME del cuerpo.
Content-Version		√	Versión del recurso.
Derived-From		√	Versión anterior.
Expires	√	√	Fecha en la que el documento queda caducado.
Last-Modified	√	√	Contiene la fecha y hora en la que el recurso fue modificado por última vez.
Link		√	Contiene información sobre los links del documento.
Title		√	Contien el título del documento.
Transfer-Encoding		√	Indica la transformación aplicada al cuerpo del mensaje, como, por ejemplo, Base64, uuencode, etc.
URL-Header		√	Contiene la parte del nombre del URL.

Tabla 3.7: Headers HTTP referentes al cuerpo del mensaje

Sintaxis	Ejemplo

<versión HTTP> <código> [<explicación>]<crlf>
<header>: <valor> <crlf>]
⋮
⋮
⋮
<crlf>
Cuerpo del mensaje]
	`HTTP/1.0 200 OK`
	`Server: Apache/1.3b3`
	`Mime-Version: 1.0`
	`Content-type: text/html`
	`Content-length: 1000`
	
	`<HTML>`
	⋮
	`</HTML>`

Tabla 3.8: Ejemplo de respuesta HTTP

- 5xx: Error del servidor

- **Los headers de respuesta** contienen información que definen al servidor y a la respuesta. Su formato es idéntico al de los headers de las peticiones, salvo que los headers en sí son distintos (se muestran en la tabla 3.6).

- **El cuerpo del mensaje** contiene la información pedida, una vez más, codificada utilizando MIME.

En la tabla se muestra la sintaxis de una respuesta HTTP y una respuesta de ejemplo.

Son incontables las referencias existentes en bibliografía y en sitios Internet. Aquí se presentan unas cuantas: [OH97], [FGM+97], [Won97], [Mue96].

3.2.2 CGI

Aunque utilizando sólo HTTP se pueden escribir aplicaciones distribuidas (como de hecho se puede hacer con sockets), el hacer que cada programa servidor que atienda a cada servicio tenga que trabajar al nivel HTTP (atendiendo a las cabeceras, mensajes y códigos de estado) resulta una pérdida de tiempo y hace que el proceso de desarrollo sea más propenso a errores. Por otro lado, hasta el momento de la introducción de CGI, los servidores HTTP existentes se limitaban a devolver los URLs especificados por los clientes. Se optó pues por desarrollar un protocolo estándar que permitiera que los servidores delegaran algún servicio a programas dedicados. A este interfaz entre los programas dedicados y los servidores estándar HTTP se le denominó CGI (*Common Gateway Interface*). Este interfaz es independiente del lenguaje de programación utilizado para desarrollar los servidores dedicados (llamados también programas CGI[††]), y comunica a estos últimos los valores de la petición del cliente a través de las variables de entorno (o *environment*) y de la entrada estándar (`stdin` para los programadores C). Después se verá el conjunto de variables de entorno y el formato de la entrada estándar utilizado para comunicarse con los servidores dedicados.

[††]Al principio, hubo cierta confusión, porque se creía que la programación CGI implicaba algún lenguaje nuevo, un entorno de programación o algo parecido.

Por su parte, los servidores dedicados generan la respuesta a la petición que le ha realizado el cliente. El servidor HTTP trata esta respuesta sólo para encaminarla de vuelta al cliente. Esto hace que los programas CGI (la mayoría de las veces) no tengan que preocuparse de detalles del protocolo HTTP.

Nótese (figura 3.8) cómo los programas servidor pueden acceder a otros recursos, como servidores de Bases de Datos a través de SQL, servidores de correo usando sockets y SMTP, etc. para llevar a cabo su función. Esto nos lleva a una configuración Cliente/Servidor de tres niveles en la que se nos permite integrar las aplicaciones y los datos existentes (lo que en la jerga se denominan *legacy systems*) con el WEB, ofreciéndolo así a todo el mundo Internet a través de los *browsers* HTML estándar.

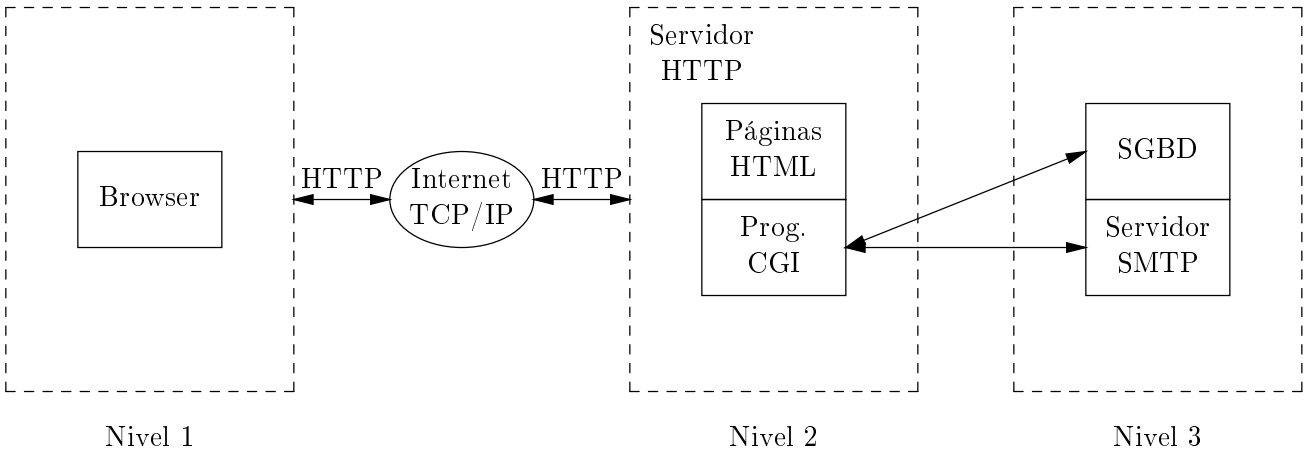

Figura 3.8: Cliente/Servidor a tres niveles utilizando HTTP/CGI.

Pero, ¿cuál es el interfaz de los clientes (*browsers*) con los programas CGI? Es decir, ¿cómo puede un cliente, desde su *browser* especificar los parámetros que reconoce un programa CGI? La respuesta está en los *forms*.

3.2.2.1 *Forms* HTML

A partir de la versión 2.0 de HTML, aparecida en 1995, se incluyó un nuevo "*tag*" en el lenguaje que permitía crear *forms*[‡‡]. Los formularios HTML son muy parecidos a los formularios en papel, con campos para rellenar, elegir, etc. Dentro de una página HTML podemos crear un formulario, que será mostrado al usuario por el *browser*, de tal manera que cuando éste pulse el botón de "aceptar" (o *submit* en inglés), la información de los campos rellenados pasará al programa CGI que especifiquemos.

Como se puede ver a simple vista, los *forms* son un interfaz sencillo y cómodo. Sin embargo, como veremos, su uso no lleva en absoluto a la consecución de programas claros, reutilizables, eficientes, etc., sino más bien a todo lo contrario. Pero primero veamos cómo todo esto se va convirtiendo en un monstruo...

En una página, un formulario se encierra entre los *tags* `<FORM>` y `</FORM>`. Éste tiene dos parámetros obligatorios: `METHOD` y `ACTION`. El método puede ser `GET` o `POST` (métodos HTTP), que especifica cómo los datos del formulario entran en el programa CGI, bien mediante

[‡‡]Lo que en español se conoce como formularios.

variables de entorno (GET), bien a través de la entrada estándar del programa CGI (GET). ACTION especifica el URL que recibirá los datos del formulario. Obsérvese la línea <FORM> del siguiente formulario HTML:

```
<html>
<form method="POST" action="http://un.servidor.es/cgi-bin/algo.cgi">
Campo de Texto: <input type="text" name="nombre" size=40 value="Diego">

<P>Area de texto: <textarea name="textarea" rows=3 cols=20>
Datos Iniciales
</textarea>
Password: <input type="password" name="password" size=10 value="asd">

<P>Radio Buttons: <input type="radio" name="myrb" value="LaTeX" checked>LaTeX
<input type="radio" name="myrb" value="troff">troff
<input type="radio" name="myrb" value="lout">lout

<BR>Check Boxes:
<input type="checkbox" name="checkbox" value="makeindex" checked>Makeindex
<input type="checkbox" name="checkbox" value="BibTeX" checked>BibTeX
<input type="checkbox" name="checkbox" value="pic" checked>pic
<input type="checkbox" name="checkbox" value="TeXCad">TeXCad

<BR>Selecci&oacute;n: <select name="so">
<option selected>Linux<option>Windows 95<option>Windows NT</select>

<p>Botones de Reset: <input type="reset">
<input type="reset" value="Borrar">

<br>Botones de Aceptar: <input type="submit">
<input type="submit" value="Aceptar">
</form>
</html>
```

El código HTML mostrado utiliza los elementos principales de los formularios. Un elemento se identifica por su nombre (la clausula "name="). A continuación (figura 3.9 en la página siguiente) se muestra en un *browser* el *form* generado.

3.2.2.2 Variables CGI

Antes dijimos que el servidor HTTP pasaba una serie de parámetros en variables de entorno y a través de la entrada estándar. También dijimos que los distintos métodos (POST, GET) indicaban la forma en que estos parámetros pasaban al programa CGI. Pues bien, los datos del formulario se envían de vuelta por el *browser* cuando el usuario pulsa uno de los botones de "submit".

La diferencia entre ambos es que en POST, los datos del formulario se facilitan al programa dedicado por la entrada estándar; utilizando GET, el programa recibe los datos del formulario

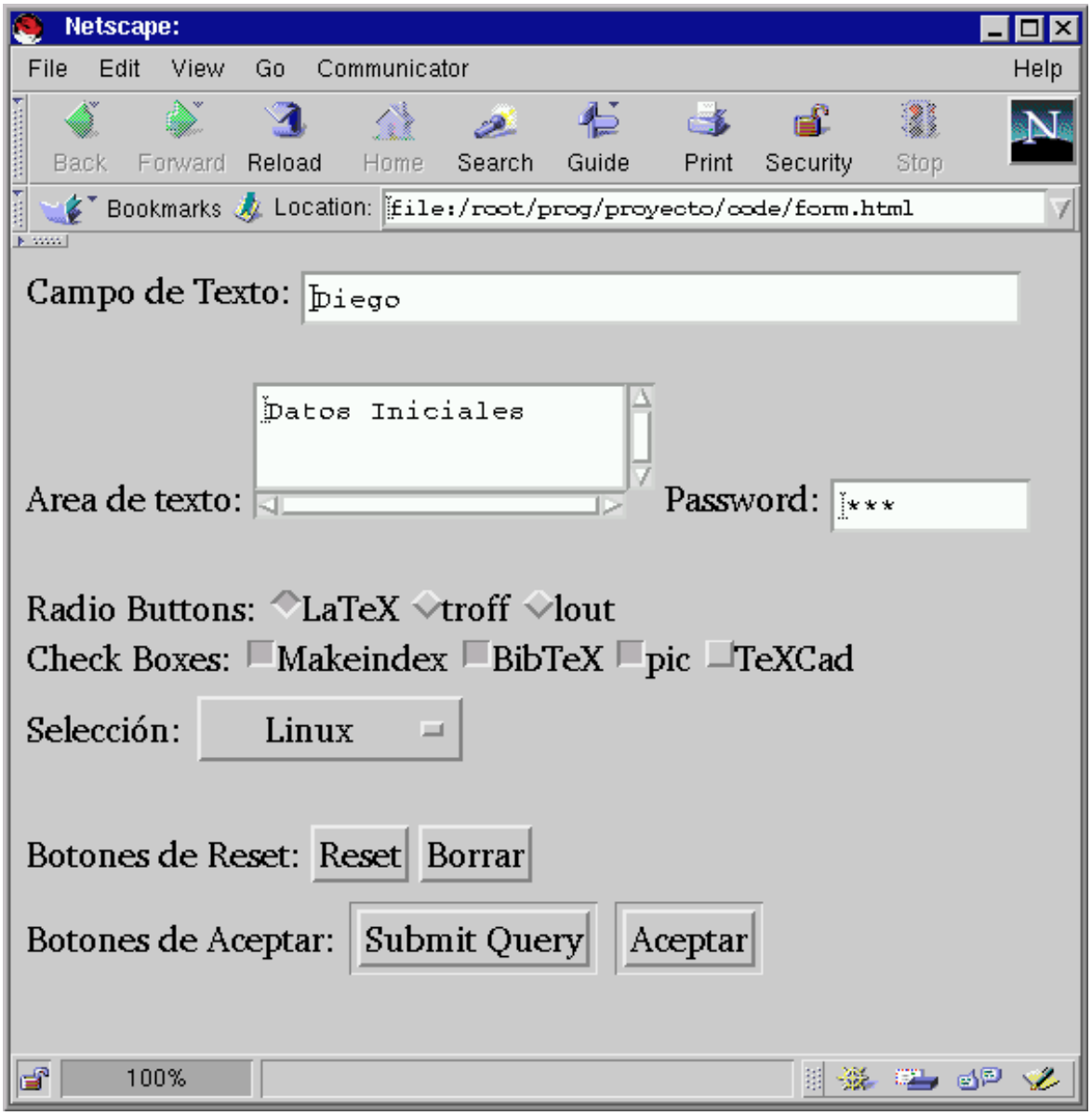

Figura 3.9: Ejemplo de *form*: elementos principales.

a través de la variable de entorno `QUERY_STRING`. Esta segunda opción plantea problemas en muchos sistemas operativos (como MS-DOS y Windows 95/NT) que tienen una limitación (absurda, por otra parte) en el tamaño de las variables de ambiente. Por lo demás, tanto en la entrada estándar como en la variable de entorno, los valores de cada campo están separados por el carácter "`&`", y se componen de pares $< nombre, valor >$, separados por "`=`". Los espacios se sustituyen por "`+`" y los "`+`" y otros caracteres especiales por "`%XX`", donde "`XX`" son dígitos hexadecimales que contienen el valor ASCII del carácter. La aplicación de ejemplo mostrará cómo manejar este flujo de variables desde la entrada estándar utilizando Perl.

En la tabla 3.9 se muestra una lista de variables de entorno CGI, tomadas en parte de [Hag96]. Estas variables de ambiente ayudan al programa CGI a sentirse a gusto en su

Variable	Valor
SERVER_ADMIN	La dirección *e-mail* de la persona que se encarga del servidor.
SERVER_SOFTWARE	Identifica al servidor y su versión.
SERVER_NAME	Identifica el nombre DNS del servidor, o, en su defecto, su dirección IP.
GATEWAY_INTERFACE	Versión del protocolo CGI utilizada, normalmente CGI/1.x.
AUTH_TYPE	Muestra el protocolo específico de autentificación.
CONTENT_LENGTH	La longitud del cuerpo del mensaje que envió el cliente. Es utilizado por el CGI para saber cuándo dejar de leer de la entrada estándar.
CONTENT_TYPE	Tipo MIME de los datos del cliente.
HTTP_REFERER	El URL del que el *script* fue invocado.
HTTP_REQUEST_METHOD	El método utilizado por el cliente.
HTTP_USER_AGENT	`User-Agent` de la petición HTTP.
QUERY_STRING	Los datos del form del usuario (si la petición es GET).
REMOTE_ADDR	Dirección IP del cliente.
REMOTE_HOST	Nombre DNS del cliente.
SCRIPT_FILENAME	El valor del *path* completo al *script* CGI.
SCRIPT_NAME	Nombre del *script*.
SERVER_PORT	Indica el puerto al que la petición del cliente se lanzó.
SERVER_PROTOCOL	Indica el protocolo que el servidor está utilizando.

Tabla 3.9: Variables de entorno CGI

ambiente.

Antes de pasar a la parte práctica de esta tecnología, veremos un escenario típico de interacción CGI. Éste nos servirá para prepararnos para la implementación de aplicación de listas de correo y para ver qué pasos necesitamos para configurar todos los elementos que permiten que la aplicación funcione.

3.2.2.3 Una interacción CGI típica

A continuación se presentan los pasos que se realizan en una interacción típica CGI:

1. **El usuario pulsa el botón "submit" del formulario**. Esto hace que el *browser* recoja la información del *form* y la ensamble en una cadena.

2. **El *browser* invoca una petición del método al servidor en el URL especificado por ACTION**. Esto hace que el *browser* cree una petición HTTP y se la envíe al servidor.

3. **El servidor inicializa las variables de entorno**. Una vez que éste recibe la petición y se da cuenta de que se pide un servicio a un servidor dedicado*, configura las variables de entorno que serán enviadas al programa CGI.

*En la siguiente sección se verá cómo.

4. **El servidor HTTP inicia el programa** CGI. El programa a ejecutar es especificado en la petición del cliente.

5. **El programa** CGI **recibe el cuerpo del mensaje**, ya sea a través de la entrada estándar o a través de la variable de entorno `QUERY_STRING`. Tiene que identificar y separar cada campo del formulario.

6. **El programa** CGI **hace su trabajo**. Típicamente, utiliza recursos de bases de datos para dar respuesta a la petición. El programa debe, entonces, dar su respuesta en HTML o en cualquier tipo MIME aceptable. (Véase figura 3.8 en la página 40).

7. **El programa** CGI **envía su respuesta a través de la salida estándar**. Ésta es capturada por el servidor WEB y es enviada al cliente añadiéndole los *headers* pertinentes si el programa no lo hizo. Si lo hizo, símplemente pasa la respuesta al cliente (los programas que rellenan sus propios *headers* son llamados *nph*, *non-parsed headers*).

8. **El servidor pasa la respuesta al *browser* del cliente**.

Como se ve, el servidor inicia los programas CGI a petición de los clientes. Esto hace la interacción mucho más dinámica. Sin embargo, nótese cómo el servidor tiene que lanzar un nuevo programa para *cada* petición. Esto será uno de los principales inconvenientes. Pero esto lo veremos después...

3.2.2.4 Alternativas a CGI: ASP, LiveWire, SSI, PHP3

CGI es muy popular y extendido en Internet. No es de extrañar que el afán de desarrollo en este campo haya traído herramientas que hagan más amigable la programación de aplicaciones basada en servidores WEB. Quizás la alternativa de mayor importancia actualmente sea *Active Server Pages*, ASP de *Microsoft*. ASP es el lenguaje de *script* para los servidores (SSI, *Server Side Includes*) de *Microsoft* (IIS, *Internet Information Server*, distribuido con Windows NT Server). Como veremos en la siguiente sección, el auge que este sistema operativo está teniendo como servidor de Internet, hace a ASP una alternativa fuerte a considerar.

La idea de ASP y de todos los SSI es no tener programas CGI separados de las páginas HTML, sino que las propias páginas se conviertan en los programas. Esto se consigue insertando el código de la aplicación en la página entre unos marcadores especiales (en el caso de ASP son "`<%`" y "`%>`"). Estos marcadores son leídos por el servidor en el momento en el que tiene que enviar la página al cliente o *browser*. Antes de enviársela, el servidor realiza la función especificada en la página, que típicamente accede a una base de datos para componer finalmente la página completa. Un ejemplo sencillo podría ser el siguiente, que iría embebido en una página ASP. El programa imprime en verde un saludo, que es diferente dependiendo de la hora del día en el que la página sea recuperada del servidor:

```
<%
If Time >=#12:00:00 AM# And Time < #12:00:00 PM# Then
    greeting = "Good Morning!"
Else
    greeting = "Hello!"
End If
%>
```

```
<FONT COLOR="GREEN">
<%= greeting %>
</FONT>
```

Como se ve, la sintaxis es muy parecida a *Visual Basic* o *VBScript*. IIS provee acceso a componentes ActiveX y a ODBC desde ASP, con lo que la programación se simplifica bastante. El único problema es que ASP es propietario.

De otro gigante, en este caso Netscape, nos viene LiveWire ó *Server-Side JavaScript* ([LH97]). Es muy parecido a ASP, solo que en este caso, se utiliza como lenguaje *JavaScript* y los separadores de código dentro de la página son <SERVER> y </SERVER>. Las características de acceso a bases de datos son similares a ASP. Además, como competencia al soporte ActiveX de los servidores de *Microsoft*, LiveWire da soporte CORBA.

Otra alternativa es la que ha adoptado el servidor Apache [APA98]. Éste incluye un módulo Perl, que es capaz de interpretar trozos de código Perl embebidos dentro de una página SSI HTML (con extensión `.shtml`).

En la misma línea está PHP3 [PHP98]. PHP3 es un lenguaje de *script* para la parte del servidor. Es parecido a ASP y LiveWire, pero es de libre distribución. Además, ofrece versiones para Windows 95 y NT, varios UNIX, etc., así como un módulo para Apache. Ofrece acceso a bases de datos Oracle, mSQL, Sybase, PostgreSQL, dBase, ODBC, dbm de Berkeley, etc. Un ejemplo que genera una página HTML con una tabla resultado de una consulta a base de datos mSQL puede ser:

```
<table>
 <tr>
  <th>Nombre del empleado</th>
  <th>Salario</th>
 </tr>

<?php

$resultado = msql($conn, "SELECT id, nombre, salario FROM Empleados");
while (list($id,$nombre,$salario) = msql_fetch_row($resultado)) {
    print(" <tr>\n",
          "  <td>$nombre</td>\n",
          "  <td>$salario</td>\n",
          " </tr>\n");
}

?>
```

3.2.3 ¿Qué elementos hay que manejar?

Es difícil resumir los elementos a manejar en la programación CGI. Y esto es así porque el 99% de los servicios actuales en Internet pasan por el Web. En primer lugar, lo que hay que configurar es un servidor Web. Este tipo de servidores, de cara a Internet, están esperando en el puerto TCP 80 (decimal), es decir, el puerto HTTP, a que algún cliente conecte con

ellos. Su principal misión es retornar páginas HTML en servicio a las peticiones de usuarios. Por lo general, los servidores tienen un directorio que se llama "*document root*" (raíz de los documentos) que actúa como raíz de toda la estructura de directorios y páginas que se ofrecen a un posible cliente. Adicionalmente, se pueden configurar directorios pertenecientes a los usuarios del servidor (como en máquinas UNIX multiusuario). En UNIX, cada usuario tiene un directorio "home", representado por la tilde ('~'). Los servidores, por lo general, aceptan que se especifique que cada usuario tendrá un directorio, llamado, por ejemplo, "public_html" donde se localizarán todos los documentos propios del usuario.

Pero como hemos visto, la labor de un servidor WEB no es sólo la de devolver documentos HTML para cada petición. Dentro de la raíz, se pueden configurar uno o varios directorios como "*script path*" (camino de *scripts* ejecutables). Estos directorios contienen programas ejecutables, es decir, programas CGI: el servidor debe seguir este protocolo para comunicarse con ellos. Por lo general, el directorio por defecto y más comunmente utilizado es el clásico "*/cgi-bin*". Hay realmente *muchos* problemas de seguridad asociados a este tipo de programas. Sobre todo cuando se considera la posibilidad de que los usuarios del *host* puedan incluir también programas ejecutables como parte de su interfaz con Internet. Sin duda la mejor referencia en este tema es "*The World Wide Web Security FAQ*", [Ste98]. El entorno UNIX es el más potente para albergar servidores WEB. Sin embargo, su uso también es más complejo. El lector puede consultar [KP87], [Fri96] y [AB94].

Para el desarrollo de la aplicación hemos elegido un entorno Windows (95 ó NT). Es indudable la importancia que este entorno está teniendo en el mundo Internet. Con esta elección, es posible presentar todos los conceptos de la tecnología CGI y a la vez, el ciclo de desarrollo utilizando la principal plataforma de soporte abierta de Bases de Datos en estos entornos: ODBC (*Open DataBase Connectivity*). Cualquiera que programe CGI's bajo Win32 debe conocerla. Nuestro programa funciona bajo Win32 utilizando ODBC, por lo que en la siguiente sección veremos cómo se pone a funcionar y se usa de una manera práctica este interfaz.

3.2.4 Un ejemplo de aplicación HTTP/CGI usando PERL y ODBC

Perl es actualmente el lenguaje más utilizado para desarrollar programas CGI. Las plataformas Windows están cada vez más presentes en Internet y están (no sin pesar por parte del autor, debido a su falta de herramientas integradas de ayuda al programador) desplazando a los servidores UNIX. El soporte estándar de bases de datos que provee Windows es ODBC... por lo que ¿cuál es la elección? Los elegimos todos.

Para ilustrar la programación CGI, vamos a desarrollar una aplicación de listas de correo o *bulletin board* en donde los usuarios pueden enviar mensajes a la lista, mensajes que serán retransmitidos a todos los usuarios que estén suscritos a la lista. El programa presenta dos interfaces: a través del WEB y a través del correo ordinario (*e-mail*). El interfaz WEB incluye un motor de búsqueda.

El esquema es sencillo. Un servidor mantiene varias listas de correo. Cada una de ellas tiene asignada una dirección de correo electrónico. Cuando un usuario envía un mensaje a esta dirección, el mensaje es automáticamente redirigido a todos los usuarios suscritos. Los mensajes también se pueden enviar a través del interfaz WEB, desde el que también se pueden realizar búsquedas por fechas o por temas.

3.2.4.1 ODBC

ODBC ([Mic95b]) es un interfaz que permite a los programadores Windows abstraerse del Sistema Gestor de Base de Datos (SGBD) específico que se esté usando, permitiendo la interacción con los datos a través del SQL de ANSI. Por lo tanto, ODBC es independiente del lenguaje, de la aplicación y del SGBD. ODBC maneja conjuntos de datos (*data sources*) a los que se asigna un nombre (*DSN*, *data source name*). Una aplicación puede crear tantos conjuntos de datos como quiera. Un conjunto de datos puede contener tablas SQL, índices, etc.

Por lo tanto, lo primero que tenemos que hacer es crear un *data source* para nuestra aplicación. Esto se puede hacer de dos maneras: bien "a mano", bien de forma automática por un programa. Si lo hacemos a mano, debemos elegir, del panel de control, el icono Win32 ODBC (figura 3.10).

Figura 3.10: Icono ODBC de 32 bits.

Si pulsamos, entramos en el administrador ODBC. Aquí debemos seleccionar la pestaña de "DSN de usuario". Debemos elegir "Agregar..." para crear un nuevo origen de datos para nuestra aplicación (figura 3.11 en la página siguiente). Una vez aquí, tenemos que elegir un *driver* para el origen de datos. Los drivers pueden ser locales o remotos (se conectan a un servidor remoto) e indican qué manejador de bajo nivel se encargará de los datos en última instancia. Aquí se nos plantean varias alternativas: elegir un formato específico (como dBASE ó ACCESS) o elegir un formato más simple y abierto (aunque en principio sea más lento), como es el driver de texto ("*Microsoft Text Driver (*.txt; *.csv)*"). Este último guarda los datos en ficheros sencillos cuyos valores están delimitados por caracteres como ';' ó ','. Elegimos este último porque nos da dos posibilidades:

- acceder desde otro lenguaje utilizando el API ODBC (como con cualquiera de los drivers), ó
- acceder a los ficheros directamente. Este tipo de ficheros son muy sencillos de manejar desde cualquier lenguaje (y más desde los lenguajes de *script*. Por su formato, se llaman CSV (*Comma Separated Values*).

Así tenemos la posibilidad—por otro lado tan acorde con el resto de este proyecto—de poder cambiar los programas eligiendo otro lenguaje de programación, aunque sea sin utilizar ODBC, o de migrar estas sencillas bases de datos a otro formato propietario más eficiente y rápido una vez que la aplicación se vaya a implantar de forma definitiva.

Al origen de datos lo llamaremos "Base de Datos de Majordomo" (*Majordomo* es el nombre que reciben los programas servidores de listas de correo). Le indicaremos también el directorio en donde se guardarán los ficheros de la base de datos (figura 3.12 en la página 49).

Una vez que aceptemos, habremos registrado en el sistema nuestro origen de datos, que podrá ser utilizado de la forma que convenga por la aplicación. Esto mismo se puede realizar de forma automática utilizando el paquete para Perl `Win32::ODBC` [ROT98] como sigue:

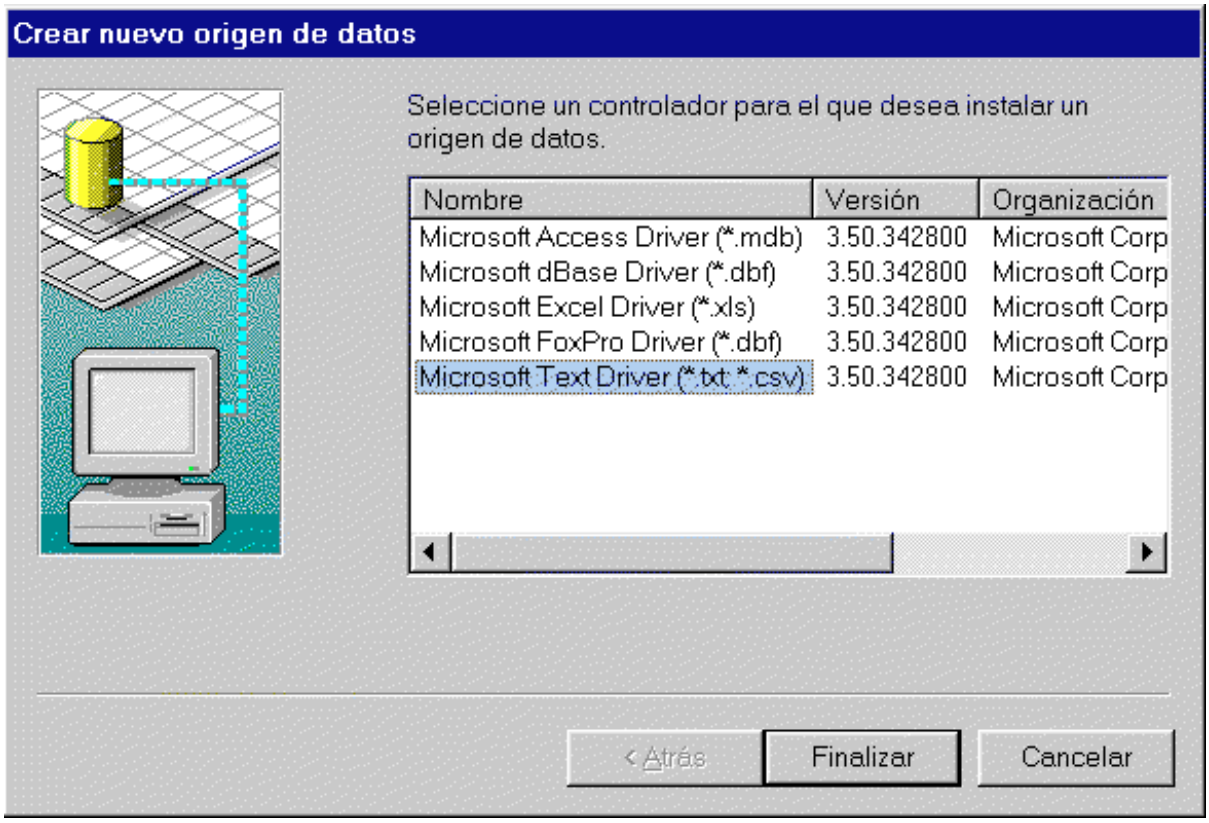

Figura 3.11: La elección del *driver*.

```
use Win32::ODBC;

#
#   Configurar el Data Source para el Majordomo
#
$myDSN = "Base de Datos de Majordomo";
$driverType = "Microsoft Text Driver (*.txt; *.csv)";
$dir = ‘cd‘;
chop $dir;
$dir .= ’\db’;

#
#   Crear el DSN
#
print "Añadiendo el DSN \"$myDSN\"...";
if (Win32::ODBC::ConfigDSN(ODBC_ADD_DSN, $driverType,
     ("DSN=$myDSN", "Description=$myDSN",
        "DEFAULTDIR=$dir", "UID=", "PWD=")))
{
    print "OK!\n";
}
```

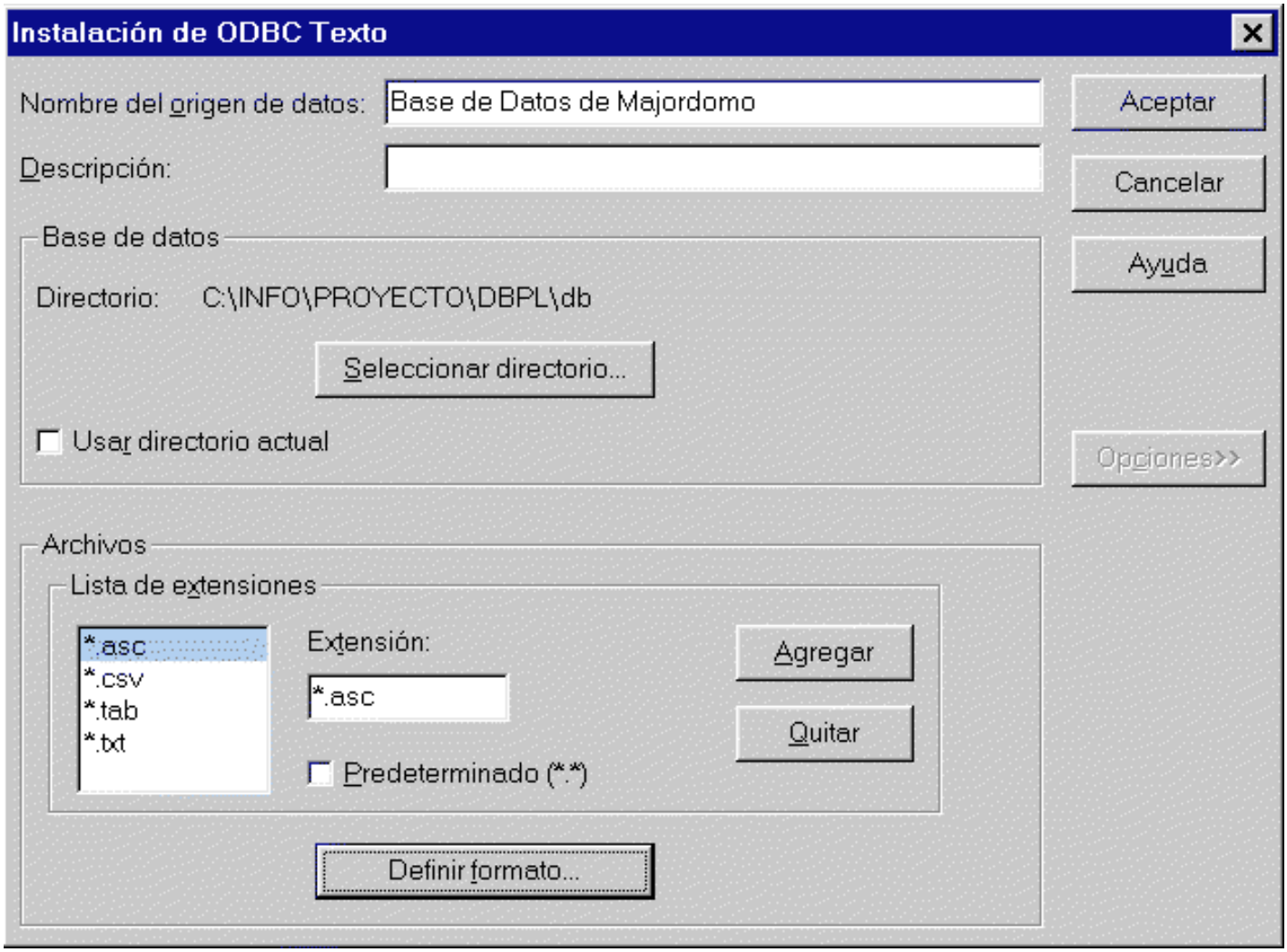

Figura 3.12: La Base de Datos de Majordomo.

```
else
{
    print "Fallo\n" . Win32::ODBC::Error();
}
```

3.2.4.2 El interfaz desde PERL

Perl para Win32, versión 5 ofrece un interfaz a través del paquete `Win32::ODBC` para manejar las primitivas ODBC a nivel del API directamente. Sin embargo, ofrece otro interfaz estándar de acceso a bases de datos utilizando SQL: el interfaz DBI/DBD ([HER98]). Utilizando este interfaz se pueden escribir las aplicaciones de acceso a base de datos independientemente del SGBD utilizado. ¿Pero esto no era lo que hacía ODBC? Sí, pero utilizando este interfaz no nos restringimos al uso de plataformas Windows: el interfaz DBI/DBD está disponible para todas las plataformas para las que está Perl (es decir, virtualmente todas). Cambiando sólo el driver DBI, podemos acceder a bases de datos Oracle, Sybase, ODBC, etc. El driver que hace

de interfaz entre DBI/DBD y ODBC está en el paquete `DBI::W32ODBC`. La organización queda como muestra la figura 3.13.

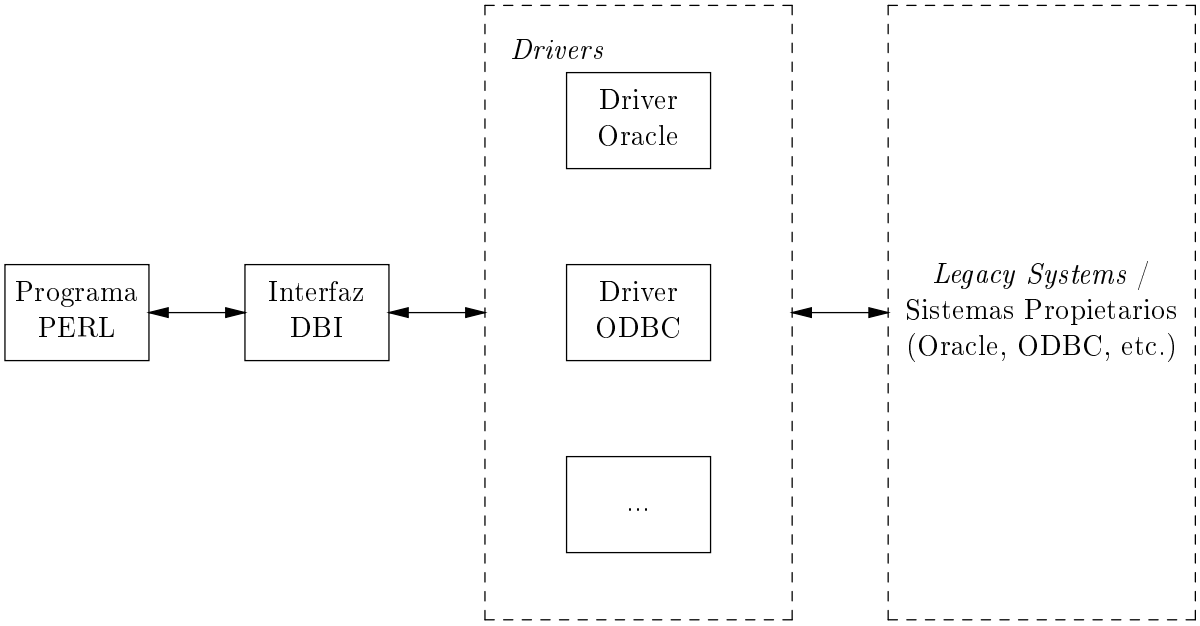

Figura 3.13: El interfaz DBI/DBD de Perl.

Lo primero que tenemos que hacer es crear las tablas. Esto se puede hacer con el siguiente programa:

```
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

#$drh = DBI->install_driver("ODBC");

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $dbh->do("drop table List");
    $dbh->do("create table List (nombre char(40),\
        email char(100),adminpasswd char(20) )");

    $dbh->do("drop table Amos");
    $dbh->do("create table Amos (email char(100),lista char (100))" );

    $dbh->do("drop table Msgs");
    $dbh->do("create table Msgs (id char(10), sent char(1),lista char(40),\
        email char(100),fecha char(20),subject char(200),msg char(255))" );

}
else
```

```
{
    print "No connect";
}
```

Se crean tres tablas: una para los datos de la lista en sí (`List`); otra para los mensajes que se envían a cada lista, (`Msgs`); y una para las suscripciones de cada lista (`Amos`). Los campos son autoexplicativos, y están expresados en SQL estándar.

Esta petición llega al *Driver* de texto de ODBC y crea los siguientes ficheros en el directorio dedicado a contener la base de datos, que, como es muestra en la figura 3.12 en la página 49, es "`c:\info\proyecto\dbpl\db`".

`List` Fichero que contiene a la tabla `List`.

`Msgs` Fichero que contiene a la lista `Msgs`.

`Amos` Fichero que contiene a la lista `Amos`.

`schema.ini` Fichero que contiene las descripciones de cada campo de las bases de datos (longitud, tipo, etc.)

Todos los listados del contenido de estos ficheros aparecen en la sección A.2. No obstante, podemos ver un ejemplo del fichero `List`:

```
"nombre";"email";"adminpasswd"
"perl";"perl";"admin"
```

Este formato, como se ve es muy simple y puede ser tratado por cualquier programa sencillo en cualquier lenguaje. Como una curiosidad, la base de datos podría incluso ser implementada utilizando *bash* y AWK [AB94], [KP87]. AWK es un sencillo lenguaje de proceso de ficheros de texto, que es capaz de dividir su entrada en campos (*fields*) de forma automática, especificando un separador de campos (*field separator*, `FS`). Para cada línea de la entrada que cumpla la condición, el programa imprimiría una entrada en la tabla en formato HTML. Por ejemplo, una consulta del tipo "`select * from List where <condición>`" podría transformarse en:

```
BEGIN {
    FS=";" # Separador de campos
}
{   if (NR != 0 && <condición>) { # La primera línea contiene los campos
        print $0
    }
}
```

Si el programa formatea su salida como una página HTML (esto es, añadiendo las líneas típicas "`Content-type: text/html`", etc.), *voilà*, ya tenemos un CGI.

3.2.4.3 La aplicación

Como toda aplicación CGI, la aplicación de lista de correo cuenta con varios programas CGI y varias páginas HTML. La aplicación queda formada por un conjunto de formularios que deben ser rellenados por el usuario y enviados al servidor, en un entorno Cliente/Servidor petición/respuesta. Las labores que realizará este programa son las de: crear nuevas listas, asignándole un nombre y una dirección *e-mail* y una clave de administración; enviar mensajes a una lista específica; suscribirse a una lista; buscar mensajes por fechas o por contenidos, responder a mensajes, etc.

En la figura 3.14 se puede ver el formulario de creación de una nueva lista de correo. En la figura 3.15 se muestra el formulario de envío de mensajes a una lista. En la figura 3.16 en la página siguiente se puede ver el formulario de búsqueda, y en la figura 3.17 se muestra una interacción en la que se responde a un mensaje encontrado a través de una búsqueda.

Figura 3.14: Formulario para la creación de una lista.

Figura 3.15: Formulario para enviar un mensaje.

Cada formulario (y, por consiguiente la página que lo contiene) está en el disco en formato HTML o bien se genera por otro programa CGI. Esto es necesario cuando los formularios contienen datos que deben ser contrastados en tiempo de ofrecérselos al usuario, como en la figura 3.15, en la que el "combobox" que guarda el nombre de la lista contiene sólo las listas definidas en este servidor. El resultado de cada form debe también ser procesado por

Figura 3.16: Formulario de búsqueda.

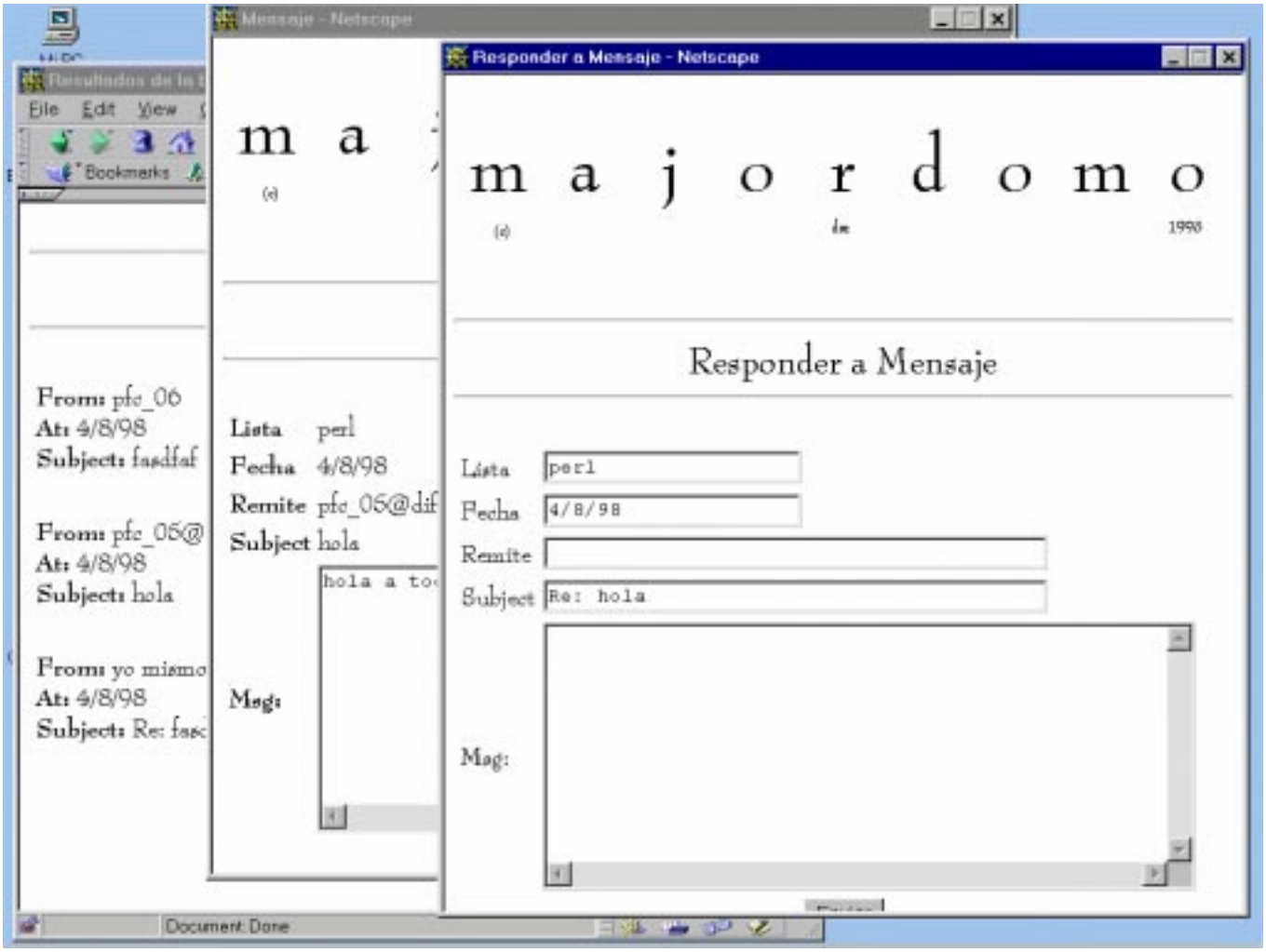

Figura 3.17: Un ejemplo de búsqueda, visualización y respuesta a mensaje.

otro programa CGI, por lo que, al final, lo normal es que existan dos programas por cada formulario uno para preprocesarlo y producir la página HTML con el formulario, y otro para

manejar los resultados del formulario. Para páginas sencillas, se puede utilizar el *tag* HTML "`<ISINDEX>`"[†].

Para ver un ejemplo representativo de un programa CGI, veremos el que genera el formulario de envío de un mensaje (figura 3.15 en la página 52). Este es el fichero `sendmsg.pl`, y genera la página HTML que se muestra en la figura:

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";

use common;
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("select * from List");
    $sth->execute;

    open(CAB,"header.html");
    $tmp = $/;
    undef $/;
    $cabecera = <CAB>;
    $cabecera =~ s/<!--HEADERINFO-->/Enviar Mensaje/g;
    $cabecera =~ s/onLoad=/onLoad=\"setdate()\"/gi;

    $javaScriptProgram="<SCRIPT languaje=\"JavaScript\">\
                    <!-- \
                    function setdate() {\
                            var now = new Date();\
                            var dia = now.getDate();\
                            var mes = now.getMonth() + 1;\
                            var anno = now.getYear(); \
                            \
                            var texto = dia + \"/\" + mes + \"/\" + anno; \
                            \
                            document.form1.fecha.value = texto; \
                    }\
                    // --> \
                    </SCRIPT> ";
```

[†]Más información en [DG96].

```
$cabecera =~ s|<!--JAVASCRIPT-->|$javaScriptProgram|g;
print $cabecera;

undef $cabecera;
close(CAB);
$/ = $tmp;

print "<P><FORM name=\"form1\"";
print "action=\"http://$host/cgi-bin/p?newmsg.pl\"";
print "method =\"POST\">";

print "<CENTER><TABLE WIDTH=\"100%\" >";
print "<TR>";
print "<TD WIDTH=\"10%\">Lista</TD>";

print "<TD><select name=\"listname\">";

#
#   Insertar las distintas listas
#
while (($nombre,$email) = $sth->fetchrow)
{
    print "<option>$nombre";
}

print "</select>";

print "</TD>";
print "</TR>";

print "<TR>";
print "<td>Fecha</td>";
print "<TD><input type=\"text\" size=20 name=\"fecha\"></TD>";
print "</TR>";

print "<TR>";
print "<td>Remite</td>";
print "<TD><input type=\"text\" size=40 name=\"remite\"></TD>";
print "</TR>";

print "<TR>";
print "<TD>Subject</TD>";

print "<TD><input type=\"text\" size=40 name=\"subject\"></TD>";
print "</TR>";

print "<TR>";
```

```
    print "<TD>Msg:</TD>";
    print "<TD>";
    print "<textarea name=\"msg\" rows=10 cols=50></textarea>";
    print "</TD>";
    print "</TR>";

    print "</TABLE>";

    print "<input type=\"submit\" value=\"Enviar\">";
    print "</form>";
    print "</center>";

    $sth->finish;
    $dbh->disconnect;

    &endHTML;
}
else
{
    &printError("Error en la base de datos");
}
```

Su código es representativo porque incluye la mayoría del los elementos presentes en la programación CGI:

- Utiliza acceso a ODBC a través del paquete `DBI::W32ODBC`.
- Genera una página que contiene un formulario.
- Utiliza uno de los lenguajes de *script* embebidos más utilizados actualmente por la comunidad Internet para hacer más activas las páginas WEB: *JavaScript*[‡], embebido en la página HTML generada para obtener la fecha actual del cliente. En otros puntos se utiliza para lanzar ventanas flotantes dependientes del *browser* para hacer más amena la lectura de mensajes (figura 3.17 en la página 53).

Se utilizan además algunos trucos que resultan muy útiles y conocidos para los programadores de CGI's. Por ejemplo, se utiliza una página de "plantilla" para el encabezado y pié de cada página HTML generada. Estas plantillas contienen identificadores que hacen que el programa pueda cambiarlas, por ejemplo, para reflejar un título distinto según el resultado del programa. En nuestro caso, se utilizan comentarios HTML (encerrados entre `<!--` y `-->`) para indicar puntos en los que el programa insertará información como una nueva función *JavaScript* que se ejecutará en el momento que el *browser* cargue la página. Este lenguaje embebido se utiliza para distribuir la funcionalidad entre cliente y servidor, relegando al cliente (*browser*) tareas sencillas, como comprobar que los datos en los formularios tienen cierto

[‡]Véase [Net98] para una referencia exhaustiva.

formato válido, etc. En el programa, la función que calcula la fecha en formato cadena de caracteres se muestra en el listado.

El programa modifica el encabezado para adaptarlo a su salida e incluir la función que se ejecutará al cargar la página. Accede a la tabla `List` para obtener todas las listas definidas en el servidor. Con ellas, rellena el *combobox* llamado `listname` del formulario generado. El código *JavaScript* que se ejecuta al principio simplemente obtiene la fecha del sistema cliente y construye una cadena que la representa. Este valor lo inserta en el elemento `fecha` del form. Obsérvese cómo el elemento `ACTION` del *tag* `form` apunta a otro CGI que procesará el valor del formulario y enviará de hecho el mensaje[§].

Si suponemos que el programa está en el subdirectorio `/cgi-bin` de nuestro.servidor.es, podremos enviar un nuevo mensaje pidiendo la página:

```
http://nuestro.servidor.es/cgi-bin/sendmsg.pl
```

ó algo distinta si utilizamos Windows 95 (véase nota al pie anterior).

El CGI que recoge los datos de este formulario (en realidad, cualquier CGI que tenga que decodificar los datos de un formulario) debe realizar una descomposición del *stream* de entrada (véase sección 3.2.2.2). El siguiente código realiza esta función y puede encontrarse, por ejemplo, al principio del `newmsg.pl`:

```
read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

@pares = split(/&/,$buffer);

foreach $par (@pares)
{
    ($nombre,$valor) = split(/=/,$par);
    $valor =~ tr/+/ /;
    $valor =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
    $FORM{$nombre} = $valor;
}
```

La primera línea lee en la variable `$buffer` el contenido de la entrada estándar, con la longitud que indica la variable de entorno `CONTENT_LENGTH`. En el array `@pares` se guardan los conjuntos de pares `nombre=valor`, que están separados por '`&`'. Para cada par, se separa el nombre y el valor. Al valor se le realizan las conversiones pertinentes '`+`' a espacios y las secuencias "`%xx`" en su carácter ASCII correspondiente. Finalmente, en el array asociativo `%FORM` tenemos una correspondencia nombre ⇒ valor, esto es, `$FORM{'nombre'} == valor`.

La aplicación completa consta de los siguientes ficheros:

- `index.html`. Página principal de la aplicación. Tiene enlaces a todos los componentes de la misma: búsqueda, creación de listas, envío de mensajes, etc.
- `header.html`. Encabezado común a todas las páginas generadas.

[§]La referencia que aparece es `p?newmsg.pl`. En un entorno UNIX hubiera valido con escribir el nombre del *script*, ya que en la primera línea indican con qué programa deben ejecutarse. Esto no ocurre en Windows 95, al que le tenemos que indicar el programa que debe ejecutar el *script*. En este caso, `p.exe`, un intérprete Perl (el interrogante separa al programa del argumento).

- `footer.html`. Pie de página común a todas las páginas generadas.
- `newlist.html`. Página de adición de una lista de correo nueva.
- `cgi-bin/addlist.pl`. Añade una nueva lista a las bases de datos.
- `cgi-bin/sendmsg.pl`. Genera la página que sirve para enviar un mensaje nuevo a una lista.
- `cgi-bin/newmsg.pl`. Inserta en la base de datos el nuevo mensaje enviado.
- `cgi-bin/subscribe.pl`. Genera la página de suscripción a una lista de correo.
- `cgi-bin/newsub.pl`. Inserta la nueva suscripción a la tabla `Amos`.
- `cgi-bin/search.pl`. Genera el formulario de búsqueda (figura 3.16 en la página 53).
- `cgi-bin/dosearch.pl`. Realiza de hecho la búsqueda y genera una página HTML con los resultados.
- `cgi-bin/show.pl`. Muestra un mensaje seleccionado.
- `cgi-bin/response.pl`. Genera un formulario de respuesta a un mensaje. Repite el "subject" y le antepone `Re:` (figura 3.17 en la página 53).
- `dispatch.pl`. Cada cierto tiempo envía a los usuarios que están suscritos a la lista los mensajes pendientes de enviar. Este programa utiliza sockets para comunicarse con todos los servidores SMTP de los suscritos.

¿No parecen muchos programas y un entramado demasiado complicado en comparación a la simplicidad de la aplicación? En la siguiente sección lo analizaremos.

3.2.5 Ventajas e inconvenientes

En las secciones anteriores se ha visto el proceso de desarrollo de la tecnología CGI. La cantidad de terminología, elementos, trucos y etcéteras no puede apartarnos de nuestro cometido inicial. Una vez que hemos aprendido todas estas técnicas, debemos ponernos críticos. No olvidemos que nuestra aspiración es el construir aplicaciones distribuidas

En primer lugar, analizaremos las **ventajas**.

- ✔ **Actualmente es el estándar *de facto* de desarrollo de aplicaciones Cliente/Servidor multinivel.** Esto hace que existan una gran cantidad de herramientas que facilitan el diseño de CGI's, que permiten diseñar formularios directamente desde bases de datos, muestran la estructura de enlaces de las páginas y facilitan el acceso a bases de datos para los programas, ya sea a través de SSI o a través de librerías. Es estándar, por lo que no presenta muchos problemas a la hora de cambiar la plataforma hardware/software en donde se ejecutan (siempre que se utilicen lenguajes de *script* portables para desarrollar los CGI's), y **todos** los servidores WEB ofrecen el mismo interfaz.

- ✔ **Permite dar a los usuarios un interfaz gráfico estándar y amigable, a través de vistosas páginas HTML y los *forms*.** Sin embargo, este interfaz es todavía muy limitado, existiendo sólo un pequeño número de controles de interacción con el usuario.
- ✔ **Ofrece un sistema de nombrado estándar y *universal*, a través de URLs.**
- ✔ **Ofrece un estándar de codificación y auto-descripción del tipo de los mensajes: MIME.** Esta codificación soluciona de alguna manera los problemas de estandarización de tipos vistos en Sockets. Sin embargo, junto con HTTP, introduce mucha sobrecarga en los mensajes, como después veremos.
- ✔ **Ofrece un marco de desarrollo rápido de aplicaciones que tienen una interacción con el usuario *relativamente sencilla*: petición/respuesta.** Junto con la primera ventaja, esta hace que CGI sea tan ampliamente usado. De hecho, todos los buscadores de Internet (*Altavista*, *Yahoo*, etc.) utilizan CGI's como interfaz hacia el usuario. Nadie piensa que los buscadores son aplicaciones sencillas. Sin embargo sí es claro que su interfaz con el usuario es bastante sencillo: un formulario más o menos complejo de petición de consulta y una lista de resultados de la búsqueda mostrada como una página WEB. Nótese que aunque esta búsqueda sea muy compleja e incluya a 500 índices de bases de datos, éste es un proceso que se realiza internamente por un programa que, en definitiva, es independiente del interfaz hacia el exterior. Aquí, el utilizar CGI's está bien: *choose your weapons to match the war*.

Pero nuestro interés es crear aplicaciones distribuidas a nivel empresarial y utilizando una tecnología que sea escalable. Si analizamos la aplicación realizada en la sección anterior, vemos que es bastante sencilla. Sin embargo, hemos utilizado ¡14 ficheros!, algunos de los cuales van en el "document root" y otros en el `/cgi-bin`. Estos y otros **inconvenientes** los veremos a continuación:

- ✘ **No existe un sistema de meta-información.** Al igual que ocurría con los sockets, aquí no existe un sistema que permita a un servidor describir y exponer a los clientes los distintos interfaces de los distintos servicios que éste ofrece. Es más, los servicios no están basados en interfaces y no hay separación entre interfaz e implementación. El desarrollo se hace en base a formularios que están íntimamente ligados a la aplicación y a la página HTML a la que pertenecen. No hay posibilidad de reutilización.
- ✘ **Los programas CGI incluyen el código que genera la página HTML mezclado con la funcionalidad propia del servidor dedicado.** Esto hace, a su vez que el código sea muy difícil de entender por terceras personas y que sea muy difícil de mantener, depurar, extender, etc. Además, el código está muy ligado al formulario que implementa (y sólo a ese) o a la página HTML a la que sirve de soporte; por lo tanto, poco reutilizable, nada modular, etc. Y se agrava cuando se utilizan técnicas como *JavaScript* embebido (véase el ejemplo de código de la sección anterior) y otros trucos que se usan en el ambiente Internet, lo cual no es una frivolidad, sino una necesidad, dadas las carencias de esta tecnología.
- ✘ **Un programa medianamente grande se convierte en totalmente inmanejable:** un conjunto considerable de ficheros de distintos tipos (HTML, *scripts*, *applets*, etc.) en distintos directorios y distintas máquinas y con relaciones de enlazado (*link*) de unos a

otros, lo cual lleva muy rápidamente a errores debido a la complejidad contra la que poco pueden hacer herramientas especializadas. No solo eso, sino que el proceso de reestructuración para una aplicación medianamente grande se hace totalmente inabordable, haciendo a esta tecnología realmente inaplicable para un desarrollo serio de aplicaciones distribuidas para el entorno empresarial.

- ✗ **Los programas son difíciles de depurar.** Cuando un CGI falla, el servidor retorna una página de error: "*500 Internal Server Error*". Esta es toda la información que tenemos sobre el error. Debemos recurrir entonces a mecanismos más o menos truculentos para depurar el programa que falla. Otras veces el programa funciona perfectamente desde la línea de comandos, pero no como un CGI... No muy atractivo.

- ✗ **El protocolo HTTP y los servidores WEB no guardan el estado entre invocaciones (*stateless*).** Se podría decir que este es *EL* gran problema. Como vimos, las peticiones HTTP son totalmente independientes unas de otras. ¿Pero qué interacción hay más simple en cualquier aplicación distribuida que el concepto de transacción? Y ¿cómo se pueden implementar transacciones con esta limitación?. Aquí empiezan las chapuzas. Básicamente hay dos maneras de implementar una cierta noción de estado para los servidores WEB:

  - Las *Cookies* de Netscape. Consisten en un conjunto de datos que se envían al *browser* para que identifique al cliente en subsiguientes peticiones. Cada *cookie* tiene una duración que habilita al usuario para realizar transacciones de una determinada duración. En cada siguiente petición, el *browser* adjunta el *cookie*. El servidor, al recibir la petición con la identificación es capaz de reconocer que mantiene una "conexión virtual" con el cliente y retomar la ejecución teniendo en cuenta esa conexión. Este es un sistema muy pobre y que ofrece problemas de privacidad.
  - Utilizar campos ocultos (*hidden*) en los formularios. Estos campos no se muestran al usuario. Pero al pulsar el botón de "aceptar" se envían al servidor con el resto de los datos. El servidor, al obtener los datos, basándose en el valor de los campos *hidden* es capaz de continuar la conexión con ese usuario en particular. Este sistema tiene todavía más problemas de privacidad. Ni que decir tiene que ambas técnicas hacen aún más engorrosa la programación CGI (aunque parezca mentira).

  Un ejemplo práctico es cuando un servidor de búsqueda (como [FAS98]) muestra sólo los 50 primeros resultados. Resultaría muy ineficiente que cuando le pidiéramos los 50 siguientes, realizara otra vez la búsqueda en la base de datos. Lo que hace entonces es guardar los resultados en un fichero temporal y enviar un identificador de ese fichero (utilizando uno de los dos métodos vistos arriba), para que cuando le pidamos los 50 siguientes, el *browser* envíe automáticamente la identificación del fichero y le permita devolvernos el resultado sin realizar la búsqueda (más costosa).

- ✗ **La codificación MIME, el protocolo HTTP y el interfaz CGI introducen mucha sobrecarga de información y de tiempo.** Mientras que en los Sockets el tiempo era una de las ventajas, para los CGI, la velocidad es uno de los mayores problemas. La siguiente tabla muestra el tiempo necesario para realizar un *ping* en CGI (la configuración de equipos es la misma que en la sección 3.1.4):

  es decir, ¡200 veces más lento que los sockets!

CGI **Local** **(entre procesos)**	CGI **Remotos** **(Ethernet a 10 Mbps.)**
600.8 milisegundos	603.8 milisegundos

Tabla 3.10: Tiempo *ping* para CGI.

- ✖ **La ejecución de un** CGI **implica el lanzamiento de un nuevo proceso (*spawning*).** El lanzamiento de un proceso es un procedimiento costoso para un Sistema Operativo: tiene que cargar en memoria el programa desde el disco, tiene que asignarle memoria propia y tiene que asignarle un tiempo de CPU para su ejecución. Pero ¿qué pasa si si se conectan simultáneamente 100 clientes? ¿Y si se conectan 1000? Obviamente, el Sistema Operativo no es capaz de atender todas estas peticiones y se colapsa. Este último inconveniente lo solucionan los *Servlets*. Los veremos en la siguiente sección.

Esta sección nos ha introducido en los conceptos de programación CGI en Internet. Cualquiera que vaya a desarrollar aplicaciones distribuidas en este entorno debe conocer estos conceptos. Sin embargo, hemos visto que para llevar a cabo una aplicación más o menos grande y seria el programador debe ser una especie de mago, arrancándole trozos de funcionalidad al cliente y al servidor utilizando *JavaScript*, *Cookies* y otros conjuros para manejar, depurar y mantener el conjunto inconexo de páginas HTML, CGI's, etc. Hacer de mago está bien para los ratos libres, pero no a nivel profesional.

3.3 Java™ *Servlets*

Los *Servlets* ([JWS97], [Sho98]) vienen arropados por la popularidad que ha alcanzado Java como lenguaje de programación para Internet. Surgieron por varias razones. Analizando un poco la situación podemos llegar a deducirlas:

- Java se ha hecho muy popular en la comunidad Internet, pero en su mayor parte como pequeñas aplicaciones embebidas en los *browsers* (*applets*) y sobre todo en la parte cliente de la interacción. El desarrollo de un servidor WEB completamente en Java (y por tanto, totalmente portable entre plataformas y Sistemas Operativos) permitiría sacar provecho de esta popularidad para permitir a este lenguaje entrar en el mercado de los servidores.

- Si el lanzamiento de un programa CGI ya es lento de por sí (véase sección 3.2.5), es aún más lento con los programas Java. ¿Por qué? Pues porque para cada CGI escrito en Java que un cliente obligue a lanzar debido a su petición se tiene que lanzar una máquina virtual Java (*Java VM*) que ejecutara los *bytecodes* Java correspondientes al servidor dedicado, y resulta que esta máquina virtual es muy intensiva en CPU, lo que hace más lento todavía al sistema.

¿Cómo se soluciona esto? Pues aprovechando de alguna manera el mecanismo de multitarea de Java (*multithreading*) para que en todo momento sólo exista un proceso del sistema (una máquina virtual Java) ejecutando tantas tareas Java como se necesiten. Esto evita tener que cargar una máquina virtual que dé servicio a cada petición del usuario: basta con crear una nueva tarea Java (*thread*). Esto es lo que hace Java Web Server (y todos los servidores que soportan *Servlets*). De esta forma permiten al programador utilizar Java para producir un

servicio a clientes que llega a ser más rápido que utilizando CGI's, aprovechando, además, las características que hicieron a Java tan famoso: independencia de la plataforma, etc.

La otra ventaja que poseen los *Servlets* sobre los CGI's normales es que ofrecen un *framework* (conjunto de clases especializadas) en el que se pueden desarrollar servidores de forma muy sencilla, derivando de clases especializadas en tratamiento de peticiones HTTP, métodos ya implementados que decodifican los argumentos de las peticiones de forma transparente al *servlet*, etc.

Sin embargo, no aportan ninguna mejora conceptual a los programas CGI's. Es por ello que esta sección no será muy larga y remitirá a la anterior a la hora de ver las ventajas e inconvenientes.

Esta sección supone un conocimiento previo de Java. Para más información, consúltese [SUN98], [Wut96] y otras muchísimas referencias existentes en Internet y en la bibliografía.

3.3.1 Introducción a los *Servlets*

Los *Servlets* se ejecutan en la máquina virtual del servidor Web en donde están registrados. Esta máquina virtual provee un conjunto de clases que ayudan al *servlet* a recibir las peticiones de los clientes y a enviarle su respuesta.

Un *Servlet* es una clase Java que implementa el *interface* `javax.servlet.Servlet`, bien directamente, o bien heredando (`extends`) de una clase especializada, como `javax.servlet.http.HttpServlet`. Este interface contiene métodos para manejar el ciclo de vida del *servlet* y para conectarlo con las peticiones de los clientes.

El ciclo de vida de un *servlet* es sencillo: Cuando tiene que atender una petición de un cliente y el *servlet* no se encuentra cargado, se carga y se inicializa a través del método `init()`. Una vez que ha sido iniciado, ya está listo para recibir peticiones de clientes. Nótese que se puede atender concurrentemente a varios clientes, por lo que el código del *servlet* debe ser reentrante (o, en inglés, *thread-safe*). Esto se consigue a través de métodos `synchronized`. Aún así, si el *servlet* no quiere ser multitarea, puede implementar el interface `javax.servlet.SingleThreadModel` para indicarlo. Este interface no implica el desarrollo de ningún método adicional. Una vez que el *servlet* ha terminado su servicio a los clientes, puede ser descargado. Antes de serlo, la máquina virtual llama a su método `destroy()`. En este método, un *servlet* puede liberar los recursos externos que esté utilizando, como conexiones a otros SGBD, etc.

Cuando un *servlet* recibe una petición de un cliente, el servidor Web llama a su método `service()` con dos objetos: un objeto que implementa el interfaz `javax.servlet.ServletRequest` que encapsula a la petición del cliente, así como a un *stream*, `javax.servlet.ServletInputStream` que le permite leer los datos de la petición del cliente; y otro objeto que implementa el interfaz `ServletResponse`, que encapsula la respuesta que el servidor dará al cliente, proporcionando también un *stream* de salida hacia el cliente: `ServletOutputStream`.

En el paquete `javax.servlet.http` se incluyen versiones especializadas de estos objetos petición y respuesta, que contienen métodos específicos de HTTP (sección 3.2.1.2), como el tipo MIME, obtener el `QUERY_STRING` en el caso de las peticiones GET, etc. Un `HttpServletRequest` permite obtener los nombres de los parámetros pasados al servidor (los nombres de los controles del formulario) utilizando el método `getParameterNames()`, que devuelve un objeto que implementa el interface `Enumeration`. Los valores de estos

parámetros se pueden saber con otra función: `getParameterValues()`, dándole como argumento el nombre del parámetro que nos interesa.

Además, los *servlets* que heredan de `HttpServlet` pueden implementar una serie de métodos que serán llamados según la petición del usuario:

- `doGet()` para las peticiones GET y HEAD.
- `doPost()` para las peticiones POST.
- `doPut()` para las peticiones PUT.
- `doDelete()` para las peticiones DELETE.

También existen los métodos `doOptions()` y `doTrace()`, para las peticiones OPTIONS y TRACE respectivamente, pero estos no se suelen redefinir, ya que proveen la funcionalidad esperada por el protocolo.

Todas estas ayudas hacen más fácil la programación de servidores especializados utilizando Java. En la sección dedicada al ejemplo veremos cómo todos estos elementos se ponen en funcionamiento.

En la siguiente sección veremos los pasos necesarios para instalar un nuevo *Servlet* en un servidor que los soporte. Para este proyecto hemos elegido *Java Web Server*.

3.3.2 ¿Qué elementos hay que manejar?

Al igual que ocurría con los CGI, hay que poner en marcha un servidor Web, pero en este caso que de soporte a *Servlets*. Actualmente, IIS (*Internet Information Server*) de Microsoft, Netscape Enterprise Server, Java Web Server (JWS) y otros soportan *Servlets*. Hemos elegido este último por ser de libre distribución. Pero, en general, los pasos que se siguen para hacer que nuestro servlet funcione son similares.

Supongamos que hemos desarollado un *Servlet*. Éste, una vez compilado (véase la sección siguiente), estará en forma de un fichero `.class`. Este fichero debemos copiarlo al directorio `/servlet` de JWS.

En primer lugar, debemos entrar en el servicio de administración del servidor. JWS ofrece un interfaz basado en *applets*. Para acceder a él, basta con acceder a la dirección del servidor y al puerto 9090:

```
http://mi.servidor.es:9090/
```

Ahí sólo tenemos que introducir el usuario y la clave de administración (por defecto "admin", "admin" respectivamente) y pulsando en "Log In" entraremos en la herramienta de manejo de servicios. De ahí, debemos elegir el servicio Web. Dentro de éste, se encuentra la configuración de los *servlets*. Esta pantalla se muestra en la figura 3.18 en la página siguiente.

Debemos pulsar en el panel izquierdo la opción "Add" (Añadir). Se nos pide entonces un nombre para el *Servlet* y la clase que lo implementa. A continuación tenemos que rellenar algunas opciones, como si queremos que el servicio se cargue al inicio, etc.

Al salir de este servicio de mantenimiento, nuestro *servlet* ya es accesible anteponiéndole el "directorio" `/servlet/`.

Que los *servlets* utilicen Java como lenguaje de programación y tecnología subyacente, nos abre la puerta a utilizar toda la tecnología que este lenguaje pone a nuestra disposición: RMI,

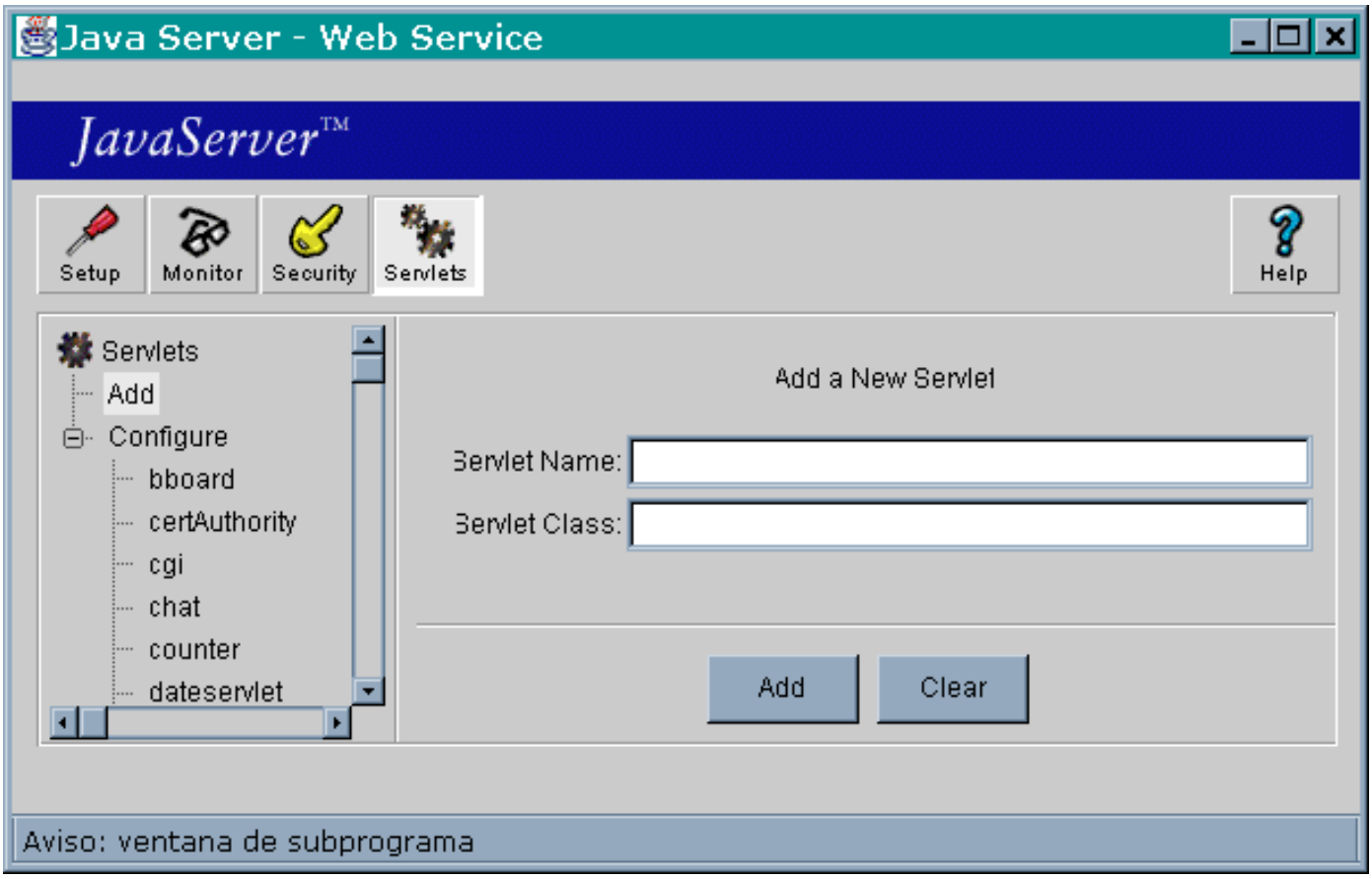

Figura 3.18: Pantalla de administración de *Servlets* del *Java Web Server*.

JDBC, etc. Es por lo tanto indispensable tener un amplio dominio de estos campos para llevar a cabo una aplicación seria utilizando *Servlets*. En la sección dedicada a RMI se explora algo más la tecnología que rodea a Java. Allí se modifica la aplicación de listas de correo utilizando RMI y ODBC a través de JDBC. En la sección siguiente se opta por modificar de forma directa los ficheros de las bases de datos. Referencias a la tecnología Java son: [SUN98], [Wut96], [OH97] y otras miles existentes a lo largo y ancho de Internet.

3.3.3 Un ejemplo de aplicación basada en *Servlets*

Debido a la semejanza de los CGI's con los *Servlets*, en esta sección sólo veremos cómo quedaría escrito como *servlet* uno de los programas que componen la aplicación de listas de correo. Concretamente, el que recibe los datos de una suscripción a una lista y actualiza la tabla `Amos` que corresponde con el programa "`newsub.pl`" de la sección 3.2.4. Como hemos utilizado ficheros CSV (sección 3.2.4.1), podemos modificar directamente el fichero que contiene la tabla. El *Servlet* se llama `SubscriptionServlet`. Para hacer que se ejecute este servidor en lugar del anterior, debemos cambiar el código del programa que generaba ese formulario: `subscribe.pl`. Ahora, el botón "submit" debe llevar la dirección del servlet:

```
<FORM METHOD="POST" ACTION=http://mi.servidor.es/servlet/SubscriptionServlet>
```

El programa se muestra a continuación:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;


public class SubscriptionServlet extends HttpServlet
{
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException
    {
        // Iniciar el tipo MIME del contenido de la respuesta
        res.setContentType("text/html");

        // Obtener el stream de comunicación con el cliente
        ServletOutputStream out = res.getOutputStream();

        // Obtener los datos del cliente. En este caso, sólo nos
        // interesan los campos 'listname' y 'email'.
        String listname = req.getParameterValues("listname")[0];
        String email = req.getParameterValues("email")[0];

        sendHeader(out,"Suscripci&oacute;n a una lista");

        try
        {
            // Añadir al fichero de suscripciones
            FileWriter outFile = new FileWriter("c:\\Amos",true);
            PrintWriter of = new PrintWriter(outFile);
            of.println("\"" + email + "\";\"" + listname + "\"");
            outFile.close();

            out.println("<CENTER>Ha quedado suscrito a la lista " +
                        listname + ".</CENTER>");

        } catch (IOException e) {
            e.printStackTrace();
            out.println("Error accediendo al fichero.");
        }

        sendFooter(out);

        // Terminar la respuesta cerrando el Writer
        out.close();
    }
```

```
    private void sendHeader(ServletOutputStream out, String title)
        throws IOException
    {
        FileReader inFileReader = new FileReader("c:\\header.html");
        BufferedReader inFile = new BufferedReader(inFileReader);

        String line;
        String headerInfo = "<!--HEADERINFO-->";
        while (null != (line = inFile.readLine() ))
        {
            int index;

            // Encontrar el identificador "<!--HEADERINFO-->"
            if (-1 != (index = line.indexOf( headerInfo )))
            {
                try {
                    String newLine =
                        line.substring(0,index) +
                        title +
                        line.substring(index + headerInfo.length(),
                                                    line.length());

                    out.println(newLine);
                } catch (Exception ignore)
                {}
            }
            else
            {
                out.println(line);
            }
        }
        inFileReader.close();
    }

    private void sendFooter(ServletOutputStream out) throws IOException
    {
        FileReader inFileReader = new FileReader("c:\\footer.html");
        BufferedReader inFile = new BufferedReader(inFileReader);

        String line;
        while (null != (line = inFile.readLine() ))
        {
            out.println(line);
        }

        inFileReader.close();
    }
```

```
    public String getServletInfo()
    {
        return "Servlet de Suscripción";
    }

}
```

Se implementa la función `doPost()`, ya que el servlet atiende a una petición POST con los datos del formulario. No tenemos que hacer la decodificación de los pares (nombre, valor), sino que están disponibles gracias al método `getParameterValues()` de la petición. El programa accede al fichero `Amos` para añadir una línea correspondiente a la suscripción actual. Se utilizan dos funciones, `sendHeader()` y `sendFooter()` para adaptar al servidor el encabezado y el pie común de las páginas HTML de la aplicación. Los *paths* de los ficheros deben sustituirse por el directorio donde se encuentren en el sistema final.

La compilación se debe realizar de tal manera que se incluya en el `CLASSPATH` las clases del servidor. Si suponemos que el servidor ha sido instalado en el directorio `<srv>`, la compilación se puede hacer modificando la variable de entorno:

```
CLASSPATH=$CLASSPATH;<srv>/lib/classes.jar
(en Windows sería "set CLASSPATH=%CLASSPATH%;<srv>/lib/classes.jar")

javac SubscriptionServlet.java -d <srv>/servlet
```

donde la opción `-d` indica el directorio donde se guardará el fichero `.class` ó bien, indicando al compilador de Java que tiene que buscar esas clases, con la opción "`-classpath <srv>/lib/classes.jar;...`". Una vez que se ha obtenido la clase compilada, se debe insertar en el servidor como se explica en la sección anterior.

3.3.4 Ventajas e inconvenientes

Las tres ventajas principales que esta tecnología introduce sobre los CGI's ya se expusieron al principio de esta sección: básicamente, una mejora en la eficiencia para situaciones de carga alta, un *framework* de clases que facilitan en gran medida la escritura de programas servidores dedicados (*Servlets*), y la posibilidad de utilizar toda la tecnología y los APIs disponibles para Java, que cada vez van siendo más extensos (JDBC, *Java Enterprise Framework*, RMI, *Beans*, etc.).

Salvo esto, las mismas ventajas e inconvenientes que se dieron para CGI en la sección 3.2.5 se pueden aplicar a los *Servlets*. Los *Servlets* tampoco son la panacea que nos llevará hacia un entorno de desarrollo de aplicaciones distribuidas cómodo a gran escala.

3.4 Java™ RMI

Con RMI entramos en el mundo de los *Objetos Distribuidos*. Desde aquí hasta el final del proyecto compararemos tres alternativas para la programación de aplicaciones distribuidas utilizando Objetos Distribuidos: RMI, DCOM y CORBA. Después de estudiar estas tecnologías, y junto con el resto del proyecto, el lector tendrá un amplio panorama de cómo se

utilizan las distintas tecnologías actuales de desarrollo de aplicaciones distribuidas y de cual aplicar en desarrollos futuros, conociendo sus ventajas e inconvenientes.

Con los Objetos Distribuidos llegamos al máximo nivel de abstracción. El objetivo es ofrecer al programador un entorno en el que las llamadas a métodos de objetos remotos (residentes en otros elementos de procesamiento de la red o en otros procesos dentro del mismo ordenador) se realicen de la misma manera que las llamadas a métodos de objetos locales (residentes en el mismo espacio de direcciones que el llamante), todo ello, como su propio nombre indica, proporcionando un entorno Orientado a Objetos con polimorfismo, ligadura dinámica, desarrollo basado en interfaces, meta-información, independencia del lenguaje de desarrollo, de la plataforma hardware/Sistema Operativo y de la localización física de los objetos, etc.

El desarrollo de aplicaciones distribuidas se facilita en extremo, gracias a las abstracciones que estas tecnologías nos ofrecen.

En esta sección veremos cómo adopta RMI la filosofía de los Objetos Distribuidos. Tras una introducción a los conceptos y a los elementos que hay que manejar, pasaremos a estudiar la adaptación que la aplicación de listas de correo sufre al implementar ciertas partes utilizando RMI y programación Orientada a Objetos con Java. Como parte de la implementación de la aplicación también introducimos aportaciones Java tan importantes y necesarias para los programadores en este lenguaje como JDBC^{TM} (*Java Database Connectivity*), el interfaz estándar que Java ofrece para el acceso a bases de datos relacionales a través de consultas SQL.

3.4.1 Introducción a RMI

RMI ([OH97, cap. 12], [SUN98], [Wut96], [Moh98], [Sho97], [Cur97]) o *Remote Method Invocation* permite a objetos Java llamar a métodos de otros objetos que están ejecutándose en otras máquinas como si fueran llamadas a objetos definidos localmente por la aplicación. Esta funcionalidad se aporta desde JDK 1.1 a través de las clases `java.rmi.*` y `java.rmi.server.*`. En toda la literatura sobre Objetos Distribuidos, el proceso por el que un objeto puede invocar métodos en un objeto remoto se divide en dos partes: la obtención de una referencia al objeto remoto y la invocación propiamente dicha.

Se pueden obtener referencias de objetos a los que previamente se les haya declarado como "remotos", esto es, que ofrecen la posibilidad de ser llamados remotamente. La definición de este tipo de objetos se apoya en los "interfaces" de Java para definir el conjunto de métodos a los que un objeto remoto puede responder. Este interface debe cumplir una serie de requisitos: debe extender el interface `java.rmi.Remote` (que no contiene ningún método adicional) y todos sus métodos deben ser declarados como posibles lanzadores de la excepción `java.rmi.RemoteException`.

```
import java.rmi.*;

public interface RemoteCollection extends java.rmi.Remote
{
    public int add(Object o) throws java.rmi.RemoteException;

    // Otros métodos...
}
```

Con esto definimos un interface remoto, `RemoteCollection`, que declara los métodos que podemos invocar remotamente en objetos que lo implementen. Cuando desarrollemos un objeto que implemente este interface (servidor), si queremos que los distintos clientes lo localicen, tenemos que darle un nombre y anotarlo en el registro. El registro RMI (*rmiregistry*) es una asociación entre nombres de objetos servidores de interfaces y referencias a esos objetos. Cualquier cliente se puede conectar a un registro RMI (que no es más que un servicio basado en sockets TCP y un puerto bien conocido, implementado como una aplicación que puede estar ejecutándose en distintas máquinas, teniendo por tanto un registro distribuido) y obtener las referencias a los objetos que desee, dado su nombre. `java.rmi` ofrece un conjunto de clases que permiten conectar y obtener referencias de un registro RMI. Estas clases se podrían denominar como el "servicio de nombres de RMI", y son `Registry`, `LocateRegistry` y `Naming`:

- El interface `Registry` permite modificar las entradas en un registro, listar su contenido. El método `bind()` o `rebind()` se utiliza para crear una asociación nombre-objeto remoto; el método `lookup()` nos devuelve la referencia del objeto asociado a un nombre, etc.
- La clase `LocateRegistry` ayuda a localizar y crear registros RMI. El método estático `getRegistry()` nos ayuda a obtener un registro de un *host* específico. El método estático `createRegistry()` crea un nuevo registro.
- La clase `Naming` nos permite obtener referencias a objetos remotos y actualizar el registro (igual que el interfaz `Registry`), pero esta vez utilizando sintaxis URL: `rmi://host:puerto/nombre`, donde `host` es la dirección del equipo a donde nos queremos conectar, y `nombre` el nombre del objeto remoto al que estamos referenciando (la parte "`rmi:`" se puede omitir).

Finalmente, la obtención de una referencia a un objeto llamado `MsgCollection` que es instancia de una clase que implementa el interfaz remoto descrito arriba, podría ser algo así:

```
try{
    RemoteCollection Msgs =
        (RemoteCollection)Naming.lookup("rmi://un.servidor.es/MsgCollection");

    // Y ya podemos invocar a los métodos del objeto remoto
    // (definidos en el interface remoto) como si fueran locales
    Msgs.add( ... );

} catch (RemoteException e)
{ ... }
```

Aquí también se muestra cómo se puede invocar a los métodos definidos en el interface remoto.

Como requisito adicional, cualquier clase que implemente un interface remoto, debe heredar (directa o indirectamente) de la clase `java.rmi.server.UnicastRemoteObject`. Un ejemplo de servidor que implemente el interface `RemoteCollection` podría ser:

```
import java.rmi.*;
import java.rmi.server.*;

public class RemoteCollectionImpl
    extends java.rmi.server.UnicastRemoteObject
    implements RemoteCollection
{
    // Constructor
    public RemoteCollectionImpl()
    {
        super();
    }

    public int add(Object o) throws RemoteException
    {
        ...
    }

    // Más métodos del interface

    // Registro del servidor
    public static void main(String args[])
    {
        // Crear e instalar un "security manager"
        System.setSecurityManager( new RMISecurityManager() );
        try {
            RemoteCollectionImpl obj = new RemoteCollectionImpl();
            Naming.rebind("//un.servidor.es/MsgCollection",obj);
        } catch (Exception e)
        { ... }
    }
}
```

En el método estático `main()` se muestra el código que sirve para registrar al servidor en el registro RMI de la máquina `un.servidor.es`. Hay que instalar un "security manager" del tipo "`RMISecurityManager`" que gestiona toda la descarga de clases necesaria y los mecanismos de seguridad asociados, como veremos más tarde.

Pero aún falta un elemento muy importante. Antes dijimos que, una vez obtenida una referencia, el cliente podía invocar métodos en el objeto como si fuera local. Sin embargo éste no lo es. Entonces ¿quién se encarga de traducir las llamadas locales a llamadas en el objeto remoto, posiblemente a través de una red? La respuesta está en el par de clases *Stub* (o *Proxy*) y *Skeleton*. Para cada clase remota, se deben crear otras dos que están íntimamente relacionadas con ella: en terminología RMI, la clase `Stub` y la clase `Skeleton`. Estas clases son generadas automáticamente por la herramienta `rmic`, disponible en las distribuciones de JDK 1.1.

La primera de ellas contiene la funcionalidad encargada de transformar las llamadas locales en llamadas al objeto remoto. Para ello, dispone de métodos para empaquetar los argumentos

del método (*argument marshaling*), conectarse con el objeto remoto y enviarle la llamada con los argumentos empaquetados. Al otro lado, es decir, en la parte del objeto remoto, debe existir otra clase (*Skeleton*) que haga justo lo contrario, esto es, que decodifique las llamadas y los argumentos (*unmarshaling*) y las traduzca a llamadas al objeto remoto real, recoja sus argumentos y devuelva el resultado al invocador. De hecho, **las referencias que retorna el registro** RMI **son referencias a objetos *Skeleton* capaces de decodificar las llamadas**.

Los objetos que son parámetros de las llamadas remotas se envían por valor. Se necesita por lo tanto un interfaz que permita convertir cualquier objeto en una secuencia de *bytes* que pueda ser transmitida por la red y que, al otro lado, pueda ser convertida de nuevo en objeto. Para esto se ha definido el interfaz denominado *Object Serialization*, nuevo en JDK 1.1. Cualquier objeto que vaya a ser serializado, debe declararse como implementador del interface `java.io.Serializable`, el cual no contiene ningún método.

Pero ¿dónde están las instancias de clases `Stub` y `Skeleton`? La instancia de la última está en el mismo espacio de direcciones que el objeto servidor remoto. La primera debe ejecutarse en el espacio del que efectúa la invocación. Lo cual nos lleva a otra pregunta: ¿cómo llega la clase *Stub* al espacio de direcciones del invocador? Nótese que el invocador efectúa su petición de objetos que implementan cierto interface, luego no es consciente de la **clase** de los objetos que **de hecho** implementa ese interface. Esto requiere que las clases *Stub* se descarguen de forma dinámica o "bajo demanda". De ahí que haya que definir, como antes hacíamos, un "security manager" que monitoriza la carga de clases y *Stubs*[¶]. De igual manera, existe una clase `RMIClassLoader` que es la que en última instancia carga las clases y los *Stubs*. Esta clase se comporta de forma distinta dependiendo de si el cliente es un *applet* o una aplicación normal:

- Si el cliente es un *applet*, las peticiones de carga de clases pasan por el *browser* que lo contiene y se traducen a peticiones HTTP al servidor WEB convencional, que debe tener disponibles las clases que el cliente pedirá.
- Si el cliente es una aplicación normal Java, las peticiones se realizan a través de *Sockets* conectando con el registro RMI que sirvió las referencias.

De una manera resumida, los pasos que se llevan a cabo para que un objeto pueda invocar métodos en un objeto remoto son los siguientes:

1. El cliente, a través de un registro RMI, obtiene una referencia a un objeto de la clase *Skeleton* que implementa el interface remoto requerido.
2. Automáticamente se inicia la descarga de la clase *Stub* correspondiente a la clase del objeto del que se recibe la referencia.
3. Se crea un objeto de la clase *Stub* descargada que actúa como objeto local (*proxy*) correspondiente al objeto remoto.
4. Las llamadas locales se realizan sobre el objeto creado de la clase *Stub*. En cada invocación, se empaquetan los argumentos y se realiza la invocación sobre el objeto *Skeleton* remoto. Esta llamada es decodificada y se traduce en invocaciones de los métodos del

[¶]Nótese que la descarga automática de clases conlleva un riesgo potencial bastante alto. Véase [OH97, cap. 12] para una discusión más detallada que excede los límites de este proyecto.

objeto que realmente implementa el interface remoto. Los resultados son enviados en sentido contrario.

Hasta aquí la teoría que rodea a RMI. En la siguiente sección veremos cómo todos estos elementos se ponen a funcionar y cómo las cuestiones teóricas se implementan en un sistema real.

3.4.2 ¿Qué elementos hay que manejar?

Una vez que tenemos claro cómo funciona RMI, debemos ver cómo las herramientas disponibles ayudan a la traducción de unos interfaces, implementaciones y clientes en una aplicación que funciona realmente.

Con cualquier distribución JDK 1.1 obtenemos todas las herramientas necesarias para desarrollar aplicaciones de Objetos Distribuidos RMI basadas en Java. El proceso de desarrollo es como sigue:

1. **Definir un interface remoto**. El servidor debe declarar sus servicios a través de un interface "remoto" que hereda del interface `java.rmi.Remote`. Todos los métodos deben ser declarados como lanzadores de la excepción `java.rmi.RemoteException`.

2. **Imlementar el interface remoto**. Se debe crear una clase que implemente este interface. Esta clase debe heredar de `java.rmi.UnicastRemoteObject`.

3. **Compilar la clase servidor**. La clase que implementa el interface debe ser compilada utilizando el compilador de java: `javac`.

4. **Ejecutar el compilador de *Stub***. Se debe ejecutar el compilador de *Stubs*, `rmic` en el fichero `.class` que implementa el interface remoto. El programa tiene los mismos argumentos que `javac`, y su sintaxis es `rmic nombreDeLaClase`. Esto generará dos ficheros: `nombreDeLaClase_Stub.class` y `nombreDeLaClase_Skel.class`.

5. **Iniciar el registro** RMI **en el servidor**. Esto crea un servidor que utiliza un *Socket* para escuchar en un puerto bien conocido las peticiones de los clientes. El registro se inicia con el programa `rmiregistry`.

6. **Crear e iniciar los objetos servidores**. Se deben cargar las clases que implementan los interfaces remotos y crear instancias de ellas. Normalmente esto se realiza en la misma clase utilizando el método `main()` (véase código del servidor en la sección anterior), aunque no es un requisito.

7. **Anotar los objetos remotos en el registro**. Se deben registrar todas las instancias de objetos servidores en el registro para que los clientes puedan encontrarlas.

8. **Escribir el código del cliente**. Los clientes son programas Java normales. Cuando lo necesitan, pueden utilizar la clase `java.rmi.Naming` para obtener una referencia a un objeto remoto.

9. **Compilar el código del cliente**. Igual que antes, se puede utilizar `javac`.

10. **Ejecutar el cliente**. Se deben cargar las clases cliente, crear instancias e iniciar su ejecución. Nótese que las clases *Stub* deben estar disponibles en directorios que permitan su descarga automática. Esto significa, en el caso de que el cliente sea un *applet*, copiar las clases bajo el "document root" del servidor WEB y respetar la estructura de directorios que se corresponde con la de los paquetes Java.

Esto lleva a que las clases que hemos escrito se comuniquen de una forma satisfactoria entre ellas para realizar la función que nos propusimos con su creación.

Es claro que no acaba aquí la cosa. En el momento que se quieran realizar aplicaciones serias con acceso a bases de datos, el programador debe conocer el interfaz JDBC, el AWT (*Abstract Windowing Toolkit*) si construye *applets* o GUIs, etc. En la sección siguiente nos acercaremos a estos puntos, ya que veremos cómo se construye un *applet* que utiliza RMI para acceder a un servidor que proporciona acceso a base de datos a través de JDBC y el puente JDBC-ODBC.

3.4.3 Un ejemplo de aplicación basada en RMI utilizando JDBC

En las secciones anteriores definimos nuestra aplicación de listas de correo en términos de servicios procedurales. A la aplicación como un *todo* se le podía pedir: "*inserta este nuevo mensaje en la lista* ZZZ *de manera que se distribuya a todos los que se han suscrito a ella*" o "*suscribe a la persona con e-mail* YYY *en la lista* ZZZ", etc. Ahora, con Java tenemos que utilizar una visión Orientada a Objetos, con lo que la aplicación se ve modificada. Estas modificaciones se verán a continuación. Posteriormente se dará una introducción a JDBC, el interfaz de Java para el acceso a bases de datos SQL. Finalmente se verá cómo utilizamos RMI para conseguir la funcionalidad distribuida de la aplicación.

3.4.3.1 Modificaciones a la aplicación de listas de correo

En esta ocasión modificaremos otra parte de la aplicación—concretamente la que se encarga de recoger un nuevo mensaje de un usuario e introducirlo en una lista determinada—para hacerla "más Orientada a Objetos". Como interfaz con el usuario, hemos elegido un *Applet* Java. Este *Applet* mostrará al usuario un interfaz que le permitirá introducir un nuevo mensaje para una lista en concreto. Las tareas de este *Applet* son las siguientes:

- Obtener el conjunto de listas de correo que se mantienen en el servidor.
- Recoger, a través de su interfaz, el mensaje del usuario.
- Insertar el mensaje en el conjunto de mensajes para la lista seleccionada por el usuario.

Por lo tanto, es claro que esta parte de la funcionalidad de la aplicación trata con cuatro abstracciones fundamentales: Las listas de correo, los mensajes, el conjunto (o colección) de listas de correo y el conjunto de mensajes. Estas dos últimas clases tienen la particularidad de que las consultas y actualizaciones sobre los conjuntos que abstraen se traducen en consultas y actualizaciones a las bases de datos ODBC que tenemos implementadas. Esto significa, entre otras cosas, que el cambio que realizamos **puede coexistir con el resto de la aplicación sin que el resto deba ser modificado en lo más mínimo**.

3.4.3.2 JDBC™ y el puente JDBC-ODBC

Cuando se habla de Java y bases de datos, es inevitable hablar de JDBC™ ([SUN98], [OH97, cap. 20]). *Java Database Connectivity* es el API de Java a bases de datos SQL. Su filosofía es parecida a la de ODBC, ofreciendo un interfaz abierto en el que se pueden conectar ("plug-in") bases de datos de terceros a través de *drivers* especializados. JDBC soporta los modelos de dos, tres y n niveles.

Desde el punto de vista del programador Java, el API está formado por un paquete: `java.sql`. En él se incluyen clases que encapsulan la comunicación con bases de datos relacionales y que permiten establecer conexiones con ellas a través de distintos *drivers* propietarios, realizar consultas y actualizaciones SQL, crear tablas, etc. El ambiente de ejecución de JDBC permite cargar varios *drivers* simultáneamente (en forma de clases Java) para atacar a bases de datos de distintos fabricantes. Las clases e interfaces más importantes de este paquete son:

- Clase `DriverManager`. Es la clase encargada de manejar el conjunto de *drivers* cargados en cierto momento, establecer y terminar conexiones con bases de datos, obtener datos del *driver* que se esté utilizando, etc.
- Interface `Connection`. Encapsula una sesión con una base de datos. Permite la creación de construcciones SQL (*statements*), llamar a procedimientos guardados en la base de datos (*stored procedures*), obtener meta-información sobre la base de datos, como tablas, información de acceso, etc.
- Interface `DatabaseMetaData`. Encapsula la información sobre una base de datos relacional (meta-información): tablas, columnas, anchos, tipos, características especiales, etc.
- Interface `Statement`. Encapsula una construcción SQL con la qe se realizará una consulta o una actualización a una base de datos. A través de sus métodos `executeQuery()` (ejecutar consulta) y `executeUpdate()` (*ejecutar actualización*) se puede actuar sobre la base de datos, obteniendo con la primera función un "conjunto resultado" (`ResultSet`).
- Interface `ResultSet`. Encapsula el resultado de una consulta a una base de datos. Sus métodos nos permiten recorrer una tabla resultado fila por fila, pudiendo obtener, para cada fila, los valores de las columnas (pertenecientes a algún tipo SQL estándar) identificadas por su nombre.

Uno de los *drivers* disponibles conecta JDBC con ODBC. Es por ello llamado el puente JDBC-ODBC (*JDBC-ODBC bridge*). Con este adaptador, tenemos la posibilidad de utilizar toda la tecnología ODBC desde Java.

En la siguiente subsección se verán ejemplos del uso de JDBC y del puente. De forma general, los pasos que hay que seguir son los siguientes:

1. Cargar la clase del *driver* específico (`sun.jdbc.odbc.JdbcOdbcDriver` en el caso del puente). Esto puede realizarse cargando la clase directamente con los métodos estáticos de la clase `Class`:

   ```
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   ```

2. Registrar el driver utilizando los métodos estáticos de `DriverManager`. Esto no es necesario para la mayoría de los *drivers* (y tampoco para el puente), ya que éstos lo hacen automáticamente. No obstante, se puede utilizar el método estático `registerDriver()`.

3. Conectar a la base de datos requerida. Si el *driver* soporta nombrado URL (como de hecho lo hace el puente), se podrá establecer la conexión utilizando un URL que identifica a la base de datos. Por ejemplo, la base de datos creada para nuestro ejemplo (con DSN "*Base de Datos de Majordomo*"), se identificaría como:

   ```
   Connection con = DriverManager.getConnection(
                                   "jdbc:odbc:Base de Datos de Majordomo");
   ```

   esto es, anteponiendo la cadena "`jdbc:odbc:`" al DSN de ODBC[‖].

4. Acceder a la Base de Datos, creando consultas y actualizaciones en forma de instrucciones SQL.

3.4.3.3 El uso de RMI

Uniendo los dos puntos anteriores, lo único que falta es adaptar las clases necesarias al uso de RMI e implementar el acceso a bases de datos utilizando JDBC desde las clases "colección". La arquitectura del sistema se puede observar en la figura 3.19.

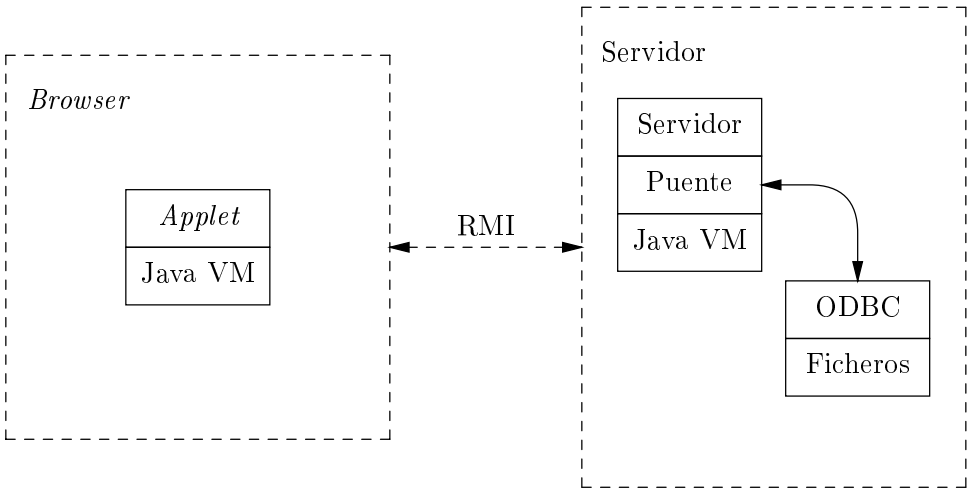

Figura 3.19: Un esquema de la aplicación utilizando RMI.

Esta arquitectura muestra que el *Applet*, a través de invocaciones remotas de métodos, accederá a las colecciones de listas de correo y mensajes. Estas colecciones están implementadas como clases que aceptan invocaciones remotas y que acceden a las bases de datos a través de JDBC para enviar de vuelta los resultados como valores de retorno de las invocaciones.

La abstracción de las listas de correo la constituye la clase `Lista`. Objetos de esta clase deberán viajar por la red como resultado de peticiones del *Applet*. Estas peticiones se harán,

[‖]Para entender mejor todas estas siglas, se recomienda ver la sección 3.2.4.1.

tanto a la colección de listas de correo como a la de mensajes, de manera secuencial, siguiendo un API parecido al del interface estándar de Java `Enumeration`. El único requisito adicional para estas clases es que implementen el interface `java.rmi.Remote`, que está vacío:

```
package majordomo;

import java.io.Serializable;

public class Lista implements Serializable
{
    private String nombre;
    private String email;
    private String adminpasswd;

    public Lista(String n,String e,String pass)
    {
        nombre = n;
        email = e;
        adminpasswd = pass;
    }

    public String getNombre() { return nombre; }
    public String getEmail() { return email; }
    public String getAdminpasswd() { return adminpasswd; }

    // Otros métodos aquí...
}
```

El interface remoto que permite ofrecer los servicios de una colección genérica de forma remota es el siguiente:

```
package majordomo;

import java.util.*;
import java.rmi.*;

public interface RemoteCollection extends java.rmi.Remote
{
    // Inserción y eliminación
    int add(Object o) throws RemoteException;
    int remove() throws RemoteException;


    // Consulta secuencial
    int first() throws RemoteException;
    Object nextElement() throws RemoteException;
    void close() throws RemoteException;
```

```
    // Otros...
}
```

Nótese cómo tanto el método `add()` como el `nextElement()` aceptan y devuelven, respectivamente, un objeto del tipo `Object`. Por la falta de parametrización en tipos del lenguaje Java, una clase que implemente este interface de forma genérica debe poseer tantas subclases como tipos distintos se vayan a insertar en las colecciones. Esta discusión atañe al lenguaje Java, y va más allá del ámbito de este proyecto. La solución presentada aquí es válida, aunque siempre discutible. La jerarquía queda como se muestra en la figura 3.20.

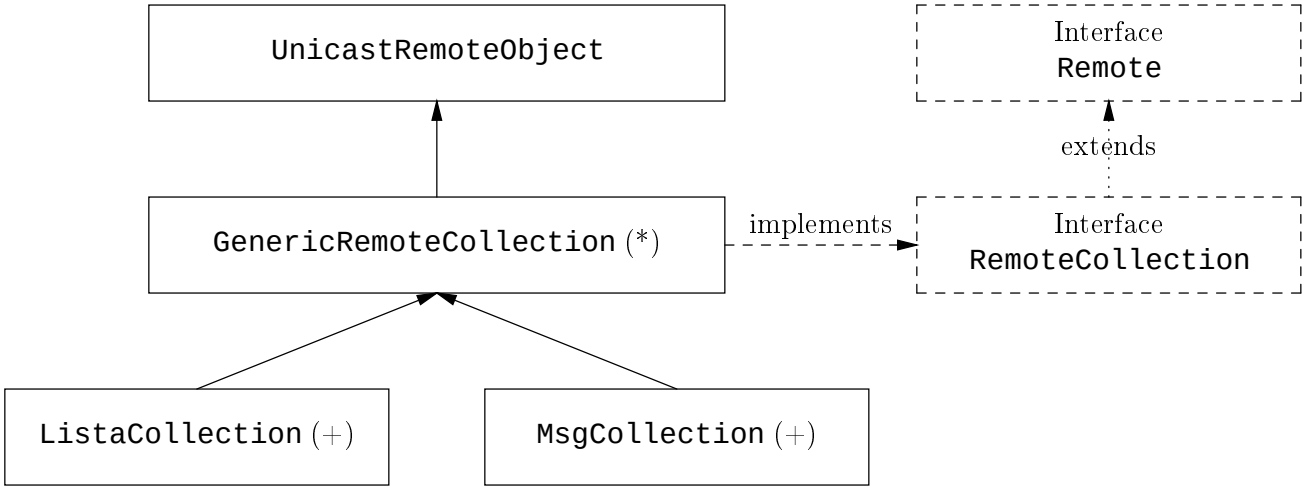

Figura 3.20: Jerarquía de colecciones con interface remoto.

La clase `GenericRemoteCollection` implementa las funciones comunes de colecciones que a su vez son remotas. Entre sus cometidos está el conectar con la base de datos. Las clases descendientes saben cómo insertar elementos de clases específicas en las tablas relacionales correspondientes. También están encargadas de devolver objetos del tipo adecuado construido a partir de los datos de una fila de la tabla relacional. A modo de ejemplo, a continuación se ofrece la clase `MsgCollection`, que hace de interfaz entre la base de datos de mensajes y la aplicación. No obstante, todos los listados de esta aplicación se pueden encontrar en la sección A.3.

```
package majordomo;

import java.sql.*;
import java.rmi.*;
import majordomo.*;

public class MsgCollection extends GenericRemoteCollection
{
    public MsgCollection(String dataSource) throws RemoteException
    {
        super(dataSource);
    }
```

```
protected String tableName() { return "Msgs"; }

public int remove() throws RemoteException
{
    // No Implementado !!!
    return 0;
}

public int add(Object o) throws RemoteException
{
    Msg tmpMsg;
    try {
        if (rs != null)
        {
            rs.close();
            st.close();
            rs = null;
        }

        tmpMsg = (Msg)o;

        st = con.createStatement();
        st.executeUpdate("insert into Msgs values ('"+
                                    tmpMsg.getId() + "','" +
                                    "N" + "','" +
                                    tmpMsg.getLista() + "','"+
                                    tmpMsg.getEmail() + "','"+
                                    tmpMsg.getFecha() + "','"+
                                    tmpMsg.getSubject() + "','"+
                                    tmpMsg.getMsg() + "')");

        st.close();
        st = null;

    } catch (SQLException e)
    {
        System.err.println("Error: "+ e);
        e.printStackTrace();
    }
    return 0;
}

public Object nextElement() throws RemoteException
{
    if (rs == null)
        return null;
```

```
        try {
            if (rs.next())
            {
                // Construir un nuevo objeto Lista y enviarlo
                return new Msg( rs.getString("id"),
                                rs.getString("sent"),
                                rs.getString("lista"),
                                rs.getString("email"),
                                rs.getString("fecha"),
                                rs.getString("subject"),
                                rs.getString("msg"));
            }
        } catch (SQLException e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }

        return null;
    }

}
```

En el código, existen unas variables de instancia que pertenecen a la clase base, `GenericRemoteCollection`, como son `con`, del tipo `Connection`, `st` del tipo `Statement`, y `rs` del tipo `ResultSet`. El método `add()` se traduce en instrucciones de inserción SQL, que son ejecutadas a través del método `executeUpdate()` de `Statement`. El método `nextElement()` devuelve un objeto del tipo `Msg` construido con los datos del siguiente elemento de la base de datos siguiendo un recorrido secuencial.

Por último, necesitamos el servidor que lance los objetos que aceptarán las peticiones y que los registre en el registro RMI.

```
package majordomo;

import majordomo.*;
import java.rmi.*;
import java.rmi.server.*;


public class Server
{
    public static void main(String args[])
    {
        System.setSecurityManager(new RMISecurityManager());

        try {
            ListaCollection lc = new ListaCollection("Base de Datos de Majordomo"
```

```
            Naming.rebind("//127.0.0.1/ListaCollection",lc);

            MsgCollection mc = new MsgCollection("Base de Datos de Majordomo");
            Naming.rebind("//127.0.0.1/MsgCollection",mc);
        } catch (Exception e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }
    }
}
```

En la parte cliente, el *Applet* realiza la misma función que su complementario form CGI.

De todo esto se puede extraer que existen muchas opciones a la hora de organizar un sistema distribuido. Las bases de datos podrían estar distribuidas entre distintos servidores, así como el registro RMI.

3.4.4 Ventajas e inconvenientes

Como ya comentamos en la introducción, con RMI nos adentramos en el mundo de los Objetos Distribuidos. Esta alternativa está claramente por encima—tanto conceptualmente como de forma práctica—de todas las tecnologías estudiadas hasta ahora. Es claro que al ver el código, la programación de la aplicación distribuida se simplifica en gran medida al poder invocar de forma transparente métodos en objetos remotos como si fueran locales, con una mínima configuración, descargando, opcionalmente, tanto los *applets* como los *stubs* de forma dinámica (o *en demanda*).

En esta sección se ha visto una tecnología que, definitivamente, es más adecuada que las anteriores mostradas hasta ahora. Sin embargo, existen otras alternativas a este nivel conceptual que ofrecen, en ciertos aspectos, unos *adenda* más completos o una perspectiva diferente. Estos son DCOM y CORBA. Las ventajas aquí expuestas se convertirán entonces en relativas al estudiar estas tecnologías.

No obstante, RMI ofrece unas **ventajas** *per se*:

- ✔ **El desarrollo de aplicaciones pequeñas es rápido y sencillo**, y no supone ningún *shock* para los programadores Java. De hecho, esta tecnología es *muy* válida para aplicaciones pequeñas de intranets corporativas que utilizan un ambiente *sólo-Java*.
- ✔ **La exposición de objetos remotos se hace a través de interfaces**, lo que facilita la modularidad, extensibilidad, reutilización, etc., además de utilizar las características de la programación Orientada a Objetos como el polimorfismo y la ligadura dinámica de una manera natural.
- ✔ **La invocación de métodos en objetos remotos es transparente al programador e independiente de la localización de esos objetos**. Lo único que se necesita es conocer el nombre de los objetos que forman parte del sistema y dónde están localizados (en qué registro RMI).
- ✔ **Los objetos pueden ser identificados por nombres y se puede utilizar un sistema de nombrado basado en URLs, mucho más sencillo y cómodo**.

- ✔ **El registro** RMI **es fácilmente configurable y permite que varios de ellos actúen en colaboración**. Además, se pueden registrar objetos remotos u objetos locales en el mismo registro, permitiendo muchas configuraciones distintas, que permiten aprovechar las características de cada equipo conectado a la red.
- ✔ **Se soporta la descarga automática de clases y *stubs* que hacen las veces de "actuadores a distancia"**, lo cual hace que:
- ✔ **La reconfiguración de los clientes es relativamente sencilla y resistente al cambio de localización de servicios y servidores**. Esto se pueden conseguir de una forma muy sencilla enviándo en las páginas HTML como parámetros a los *applets* que contienen dónde tienen que buscar un registro en concreto, o una lista de registros, etc., que puede ser modificada cada vez que el cliente pide el *applet* a través de su *browser*.
- ✔ **Favorece los clientes *pequeños* ó *livianos* (*thin clients*)** y la estandarización de interfaces de usuario a través de *applets* y el *browser universal* (lo que se ha venido en llamar *Network Computer*).

En cuanto a los inconvenientes, si comparamos esta tecnología con las características que expusimos en el capítulo 2 como óptimas o definitivamente deseables para facilitar el desarrollo de aplicaciones distribuidas, podemos estudiar los siguientes **inconvenientes**:

- ✘ **No se provee un mecanismo de meta-información y auto-descripción**. Los clientes no pueden "despertar" en un ambiente que se describa a sí mismo. Por ejemplo, que indiquen qué registros hay disponibles y cuáles son sus direcciones, que permita obtener los interfaces existentes en cada registro (no sólo los nombres de los interfaces, sino, además, qué métodos soportan, qué argumentos, etc.). Después veremos cómo CORBA si provee esta funcionalidad a través del ORB (*Object Request Broker*).
- ✘ **Sólo se soporta un lenguaje, Java, lo que hace difícil la integración de software ya construído (*legacy*)**. El sistema queda ligado a su implementación utilizando Java y ligado, como RMI, a las interioridades de este lenguaje (sistema de tipos, serialización etc.). Por ello, no posee un interfaz estándar de comunicación a través del cable, y no es compatible con el estándar CORBA: IIOP (*Internet Inter-ORB Protocol*).
- ✘ **Los objetos son pasados por valor, lo cual hace que el sistema no sea escalable a medida que el tamaño de los objetos aumenta (el sistema se hace más ineficiente en tiempo)**. La siguiente tabla muestra los resultados obtenidos en [OH97] para un sencillo *ping* con la configuración mostrada en la sección 3.1.4:

RMI **Local** **(entre procesos)**	RMI **Remoto** **(Ethernet a 10 Mbps.)**
5.5 milisegundos	5.1 milisegundos

Tabla 3.11: Tiempo *ping* para RMI.

Y esto es sólo para un *ping*. En el momento en el que la invocación pase como parámetro una colección, todos los objetos que contiene tienen que ser *serializados* y enviados por el cable. Contra esto se debe argumentar que un buen diseño tendrá en cuenta estas restricciones para alcanzar la mayor eficiencia posible.

- ✘ **No existe protocolo de paso por *firewall*.** En la mayoría de las organizaciones, un *firewall* corta las comunicaciones que no pasan por el puerto 80 (HTTP), 21 (FTP) y 25 (Mail) (estos puertos por regla general, claro). No hay un estándar que permita que las peticiones RMI pasen a través de estos puertos, y que, además, permitan a las aplicaciones que tradicionalmente están en estas direcciones (servidores WEB, *browsers*, etc.) funcionar sin dificultad.

- ✘ **No favorece la programación totalmente distribuida, sino que los roles Cliente y Servidor deben ser bien establecidos al principio**. El hecho de que los objetos se pasen por valor hace que los subsiguientes métodos invocados en esos objetos se realicen de forma local al que los recibió. Esto hace que el sistema no sea (o no se acerque a ser) *peer-to-peer*, pero en el sentido amplio de que un cliente pueda hacer en cierto momento las veces de servidor sin ningún truco extraño. Como veremos, CORBA ofrece la posibilidad de *callbacks*, que hacen que los objetos cliente se puedan comportar como servidor en un momento dado.

3.5 Java™ & DCOM

La inclusión de *Microsoft* en el mundo de los Objetos Distribuidos es DCOM ([Mic96], [Mic95a], [CHY+97], [OH97, cap. 14], [Bro94]) o *Distributed Component Object Model*[**]. Parece extraño que consideremos aquí una solución propietaria[††]. Sin embargo, el propietario en cuestión es el creador de los dos Sistemas Operativos más en auge actualmente: Windows 95 y Windows NT. Aparte de su popularidad, DCOM ha demostrado no estar todavía preparado para un desarrollo serio de aplicaciones distribuidas a nivel empresarial. A los autores Orfali y Harkey les llevó tres semanas hacer que funcionara un ejemplo simple como el que aquí se presenta, y esto contando con ayuda directa de *Microsoft*. Aún así, no pudieron conseguir estadísticas de tiempo *ping* para un servidor local (cliente y servidor en la misma máquina). En esta seción se comienza con un pequeño tutorial de DCOM y se describe la (mala) experiencia que el autor ha tenido con esta tecnología. Como ejemplo de aplicación, se implementa una pequeña aplicación "Contador" que además nos sirve para evaluar el tiempo medio de respuesta de una invocación remota de un método.

3.5.1 Una muy pequeña introducción a DCOM con Java

Como tecnología de Objetos Distribuidos que es, DCOM trabaja en torno a interfaces. Un interfaz es la definición de un conjunto de operaciones (o métodos) que un objeto ofrece hacia el exterior. Estos interfaces son independientes del lenguaje utilizado para implementar su funcionalidad. Esto, como se ve, equivale a los interfaces Java o a los ficheros "`.h`" de C++. Estos interfaces se especifican utilizando el IDL (*Interface Definition Languaje*) de *Microsoft*, que, además, no es compatible con el IDL de CORBA.

[**] Aunque por cuestiones de organización se estudia antes DCOM que CORBA, quizá convenga que el lector lea primero el capítulo 4 para una introducción sobre los conceptos que rodean a la tecnología de Objetos Distribuidos.

[††] Aunque *Microsoft* ha enviado un RFC informativo que se puede obtener en http://ds1.internic.net/internet-drafts/draft-brown-dcom-v1-spec-00.txt.

3.5.1.1 El modelo de objetos

El modelo de objetos COM[‡‡] no es un modelo de objetos clásico (al estilo de los lenguajes Orientados a Objetos tradicionales). Las tres variaciones más importantes son las siguientes:

- Los objetos no tienen identificador propio, es decir, no existe el clásico OID (*Object Identifier*) de los modelos de objetos clásicos. En su lugar, son los *interfaces* los que presentan identificadores únicos universalmente (IID, *Interface ID*, o GUID (*Globally Unique Identifier*) de Interfaz). Esto tiene algunas complicaciones, ya que, si por algún fallo en la red se pierde la conexión con un objeto servidor, es **imposible** volverse a conectar al mismo objeto, sino que el sistema puede reconectar a *cualquier otro* objeto que implemente el interfaz requerido. La representación que se hace de esto en la literatura DCOM se puede ver en la figura 3.21.

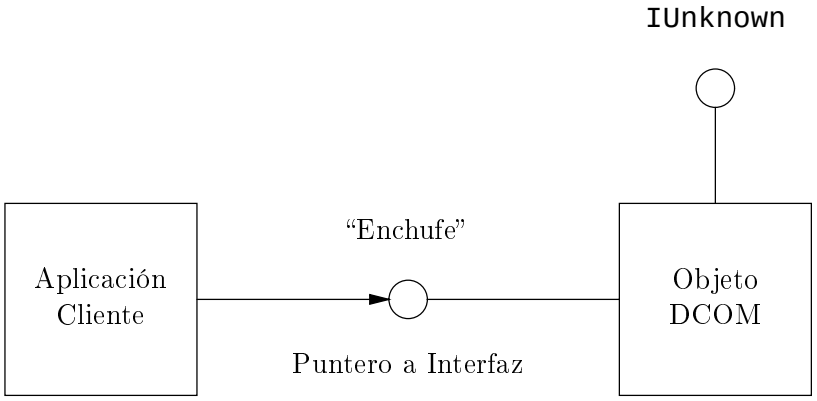

Figura 3.21: La representación de un interfaz DCOM.

- Una clase DCOM implementa una serie de interfaces, debiendo implementar obligatoriamente el interfaz ubícuo `IUnknown` (esto es, *hereda* de `IUnknown`). Un objeto DCOM es una instancia de una clase DCOM. Sin embargo, por la ausencia de identificadores de objetos, los términos de clase y objeto se funden, utilizando este último para designar a ambos. Las clases DCOM tienen también identificadores universales (CLSID).
- DCOM no soporta herencia múltiple de interfaces. Sin embargo, un componente DCOM puede implementar varios interfaces y encapsular a otros componentes.

Los servidores DCOM se implementan utilizando programas *EXE* ó *DLL* de Windows que se cargan bajo demanda cuando un cliente necesita un objeto de las clases que implementa un servidor. Por lo tanto, aquí la palabra "servidor" también tiene una connotación diferente, refiriéndose al "agente servidor" que es capaz de crear objetos para ofrecer servicio a los clientes que lo necesiten. Para que el sistema sepa cuándo debe cargar qué fichero, en el registro de Windows, bajo la clave `HKEY_CLASSES_ROOT\CLSID` debe haber una asociación entre el CLSID y el programa que contiene su implementación. Los servidores deben proveer un

[‡‡]DCOM es símplemente un estándar de interoperatividad binaria, por lo que sólo indica cómo se realiza la *transmisión* de las peticiones COM a través de distintos protocolos de transporte, como, por ejemplo, TCP/IP u otros de aplicación como HTTP. Así, tendríamos que hablar de interfaces y objetos COM.

interfaz `IClassFactory` para cada clase que implementen (para cada CLSID). Este interfaz permite crear objetos de cada una de las clases.

Como vimos, cualquier clase debe implementar el interfaz `IUnknown`. Este interfaz permite controlar el ciclo de vida de cada objeto DCOM. Los métodos que implementa son los siguientes:

`QueryInterface()` Cuando un cliente obtiene el acceso a un objeto DCOM, este obtiene un puntero a un interfaz. A través de esta función, un cliente puede interrogar a un objeto si implementa cierto interfaz (definido por su IID).

`AddRef()` Este método es llamado cada vez que se crea una nueva referencia al objeto en cuestión. DCOM utiliza cuenta de referencias para mantener la recolección automática de basura (*automatic garbage collection*).

`Release()` Este método es el inverso al anterior. Su llamada indica que hay una referencia a ese objeto que ya no se utiliza.

Por su parte, el interfaz `IClassFactory` ofrece dos métodos que ayudan a la creación de objetos de una clase:

`CreateInstance()` Crea una nueva instancia sin inicializar de la clase.

`LockServer()` Indica que el servidor debe permanecer en memoria incluso si no existen actualmente referencias a ningún objeto de las clases que implemente. Esto permite un aumento de eficiencia en algunos casos, como entornos de respuesta en tiempo real, etc.

3.5.1.2 IDL de DCOM e Invocación Dinámica

Una vez que se ha escrito un interfaz utilizando IDL, el compilador de IDL (MIDL), al igual que ocurría con RMI, genera los *stubs* del cliente y del servidor, que, entre otras cosas, realizan la decodificación de los argumentos (*marshaling*). A través de este lenguaje, DCOM ofrece un sistema de meta-información e invocación dinámica.

La librería de tipos DCOM (DCOM *Type Library*) mantiene un repositorio de los interfaces que están disponibles. Esta librería permite que los clientes construyan las llamadas de forma dinámica, esto es, en ejecución. Son capaces de obtener todos los métodos y atributos de un interfaz e invocar llamadas conforme se necesiten. Esta invocación dinámica, en terminología DCOM recibe el nombre de *OLE Automation* o símplemente *automation*.

Desde la parte servidor, un servidor DCOM que permita una invocación dinámica debe implementar el interfaz `IDispatch`. Este interfaz requiere de la implementación de una serie de métodos que ayudan a identificar, construir e invocar las llamadas construidas dinámicamente.

3.5.1.3 DCOM y Java

La integración de DCOM y Java que *Microsoft* ha preparado para su compilador, *Microsoft Visual J++*, permite abstraernos de algunas de las tareas más repetitivas del desarrollo basado en componentes DCOM. Por ejemplo, no se deben proveer factorías de clases o "*Class Factories*" a través del interfaz `IClassFactory`. Además, no es necesario que las clases Java que implementan objetos DCOM se encuentren en ningún ejecutable, sino que, ante una petición, se lanza el intérprete Java para que ejecute el servidor dedicado.

Permite así que tanto el cliente como el servidor se preocupen sólo de implementar los métodos requeridos. El problema real estará en la configuración de todo el sistema. En este punto, los creadores de DCOM no estuvieron muy finos. Programar en un entorno heterogéneo de Objetos Distribuidos requiere que todos los componentes del sistema tengan una versatilidad tal que permita a los desarrolladores elegir la configuración más adecuada. Si en un caso tan sencillo como el de la aplicación presentada aquí esto se convierte en una auténtica pesadilla, ¿qué pasará en un sistema real, donde hay que configurar cientos de componentes que interactúan dinámicamente?

3.5.2 ¿Qué elementos hay que manejar?

La programación con DCOM implica un amplio conocimiento de las interioridades de los sistemas Windows. Este es el principal problema. Programar para Windows es difícil y engorroso, basado en API's funcionales; y DCOM, con sus ligaduras para Java, no oculta esa complejidad. Por supuesto que la programación Windows a nivel de sistema queda fuera del alcance de este proyecto; no obstante, si consideramos sólo el proceso de desarrollo basado en DCOM, los siguientes puntos nos dan una idea de los pasos que hay que seguir para hacer que nuestro programa culmine "con éxito" (estos puntos se verán complementados por el desarrollo real llevado a cabo en la siguiente sección):

1. **Crear el** IDL **de** DCOM **para nuestros objetos**. Se deben definir todos los interfaces de la clase. A la clase hay que ponerle el atributo `oleautomation`.

2. **Generar GUIDs para nuestros interfaces**. Se deben generar GUIDs para cada interfaz y cada clase que declaremos en el fichero IDL. Se puede utilizar la herramienta *Microsoft Developer Studio* que genera identificadores (bajo el menú *Tools*). Estos identificadores son generados mediante un algoritmo que asegura que la probabilidad de que existan dos identificadores iguales es prácticamente inexistente (véase figura 3.22 en la página siguiente).

3. **Crear el fichero de librería de tipos (*type library*)**. Se debe aplicar el programa MIDL 3.0 o superior a los ficheros IDL escritos para generar el fichero *.TLB* para las clases.

   Sin embargo...

   Aunque la documentación de *Visual J++* dice que se puede utilizar ODL junto con el programa MKTYPLIB, este programa (ni el ODL) funcionan para invocaciones remotas DCOM. En un principio, existían dos lenguajes de descripción de objetos: ODL para describir meta-información; e IDL para describir clases e interfaces. En 1996, *Microsoft* anunció a ODL como obsoleto, y utilizó IDL para ambas tareas. El problema es que se creó la nueva versión de MIDL, la 3.0, que **sólo** se distribuye con el *Win32 SDK* ó *Visual C++*.

4. **Crear los *wrappers* Java para la clase** DCOM. La utilidad JavaTLB genera, a partir del fichero *.TLB*, los ficheros `.class` para cada interfaz. Estos ficheros hacen de *stubs*

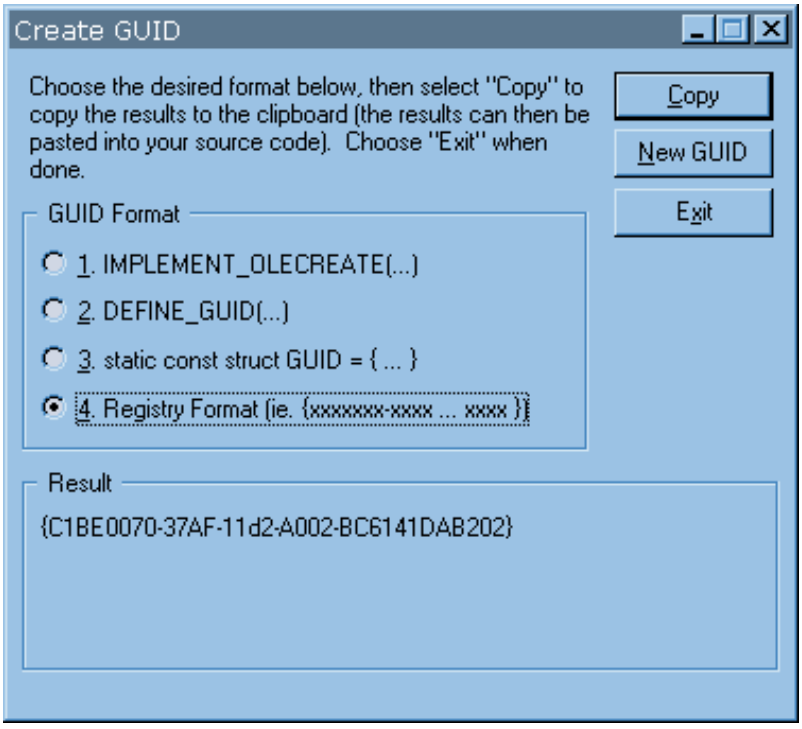

Figura 3.22: Generación de GUIDs a través de *Microsoft Developer Studio*.

como los vistos en RMI (en la siguiente sección se verán los detalles para un ejemplo real).

5. **Implementar las clases** DCOM **en Java**. Se deben crear las clases Java que implementan los interfaces definidos en IDL.

6. **Compilar la implementación**. Se deben compilar los ficheros `.java` creados usando el compilador jvc, el compilador de Java que viene con *Visual J++*.

7. **Registrar las clases Java**. Se debe utilizar la herramienta JavaReg para registrar las clases Java como clases DCOM. Éste crea las entradas pertinentes en el registro de Windows. Como desarrollador, al distribuir nuestras clases, debemos ocuparnos de que en la instalación del programa, estas clases se registren en la máquina en la que se va a ejecutar el programa.

8. **Escribir el código del cliente**. El código del cliente es código Java normal. La única restricción es que no se utilicen objetos de la clase DCOM directamente. En su lugar, se debe hacer un *casting* (o *narrowing* en Java) hacia el interfaz que queremos en realidad. El intérprete específico de *Microsoft*, jview hará toda la magia que rodea a la elección del interface correspondiente (recuérdense las peticiones `IUnknown::QueryInterface()`).

9. **Compilar las clases cliente**. Esto se hace de forma normal utilizando el compilador jvc.

10. **Registrar el cliente**. Teóricamente, sólo un simple fichero que se puede transmitir por la red (HTTP, etc.) debería ser necesario para que el cliente pueda acceder a objetos remotos. Pero actualmente, los pasos que hay que realizar son los siguientes:

    (a) Copiar el IDL al cliente.

    (b) Ejecutar MIDL.

    (c) Ejecutar JavaTLB.

    (d) Ejecutar JavaReg.

    (e) Ejecutar DCOMCNFG (el configurador de DCOM) para habilitar DCOM en ambos, cliente y servidor (este proceso se verá en la siguiente sección).

    (f) Decir a DCOMCNFG el nombre y la localización de la clase remota.

    (g) Ejecutar regedit para borrar del registro Windows las entradas `InprocServer32` ó `LocalServer32` existentes para esa clase. Como DCOM no soporta un directorio distribuido, esta configuración debe realizarse en cada máquina cliente.

11. **Iniciar el cliente**. Ejecutar el intérprete jview con la aplicación cliente. Nótese que no se tienen que inicializar el servidor: de esto se encarga el sistema (milagrosamente).

¡Y esto hay que hacerlo para cada clase! Entre otras cosas, en la configuración es en donde DCOM realmente no está a la altura de un desarrollo serio de aplicaciones distribuidas en el ambiente empresarial.

3.5.3 Un ejemplo de aplicación basada en Java™ y DCOM

La aplicación que veremos en esta sección se muy sencilla. Se trata de una clase que implementa un contador simple. Provee métodos para inicializar, incrementar y leer el contador. El programa cliente utilizará esta clase para realizar un total de 1000 incrementos y medir el tiempo medio de llamada. Esto nos servirá para establecer un tiempo *ping* para esta tecnología.

En primer lugar, veremos la definición en IDL de DCOM de este simple interfaz, llamado `ICount`. Nótese que todo elemento de esta definición tiene un GUID: la librería, cada interfaz y la clase:

```
// Count.idl

[
        uuid(619AF200-2174-11d2-9FFA-C68E0ECBEF02),
        version(1.0)
]
library Counter
{
        [
        uuid(619AF201-2174-11d2-9FFA-C68E0ECBEF02),
        oleautomation
        ]

        interface ICount: IUnknown
```

```
        {
                HRESULT set_sum([in] int val);
                HRESULT get_sum([out, retval] int* retval);
                HRESULT increment([out, retval] int* retval);
        };

        importlib("stdole32.tlb");
        [
                uuid(619AF202-2174-11d2-9FFA-C68E0ECBEF02),
        ]
        coclass Count
        {
                [default] interface ICount;
        };
};
```

Nótese que se genera la clase `Count` que implementa sólo un interfaz: `ICount`.

Seguidamente, como se vió, se debe ejecutar el programa MIDL 3.0:

```
midl count.idl
```

Esto genera el fichero `count.tlb`. Este, a su vez, debe ser pasado a través del programa JavaTLB para generar los *stubs* para Java:

```
javatlb /U:T count.tlb
```

lo que genera tres ficheros: `summary.txt`, que indica las clases que se deben implementar, el fichero `Count.class` (bytecodes Java de la clase DCOM) y el fichero `ICount.class` (bytecodes Java el interfaz). Estos ficheros se introducen en el directorio `C:\<windows>\java\trustlib\count`, donde `<windows>` indica el directorio donde Windows 95 ó NT están instalados.

Una vez que tenemos los esqueletos, debemos implementar el cliente y el servidor. El cliente

```
// CountDCOMClient.java

import count.*;

class CountDComClient
{
        public static void main(String args[])
        {
                try{
                        // Encontrar el objeto con interfaz ICount
                        System.out.println("Creando el objeto Count");
                        ICount counter = (ICount)new count.Count();

                        // Establecer "sum" al valor inicial de 0
```

```
                        System.out.println("Inicializando sum a 0");
                        counter.set_sum((int)0);

                        // Calcular el tiempo inicial
                        long startTime = System.currentTimeMillis();

                        // Incrementar 1000 veces
                        System.out.println("Incrementando");
                        for (int i=0; i< 1000; i++)
                        {
                                counter.increment();
                        }

                        // Calcular el tiempo final
                        long stopTime = System.currentTimeMillis();
                        System.out.println("ping = " +
                                ((stopTime - startTime)/1000f)
                                +  " msecs");
                        System.out.println("Sum = " + counter.get_sum());
                } catch(Exception e)
                {       System.err.println("Error:");
                        e.printStackTrace();
                }
        }

}
```

El servidor por su parte, queda en una clase que implementa el interfaz `ICount`. Importa los paquetes de COM de *Microsoft* (`com.ms.com`):

```
// Count.java

import com.ms.com.*;
import count.*;

class Count implements ICount
{
        private int sum;

        public int get_sum() throws ComException {
        return sum;
        }

        public void set_sum(int val) throws ComException {
        sum = val;
        }
```

```
        public int increment() throws ComException {
        sum++;
        return sum;
        }
}
```

en donde todos los métodos son declarados como posibles lanzadores de la excepción `ComException`. El siguiente paso es compilar y registrar la clase servidor:

```
jvc -d c:\<windows>\java\trustlib Count.java
```

El registro se lleva a cabo con javareg:

```
javareg /register /class:Count
    /clsid:{619AF202-2174-11d2-9FFA-C68E0ECBEF02} /surrogate
```

(nótese cómo el CLSID es el que se encuentra justo encima de la declaración `coclass` en el fichero IDL, declaración que identifica a la clase en sí).

Finalmente, y antes de ejecutar el cliente con `jview CountDCOMClient`, hay que configurar debidamente el subsistema DCOM, como ya dijimos.

En primer lugar, debemos borrar las entradas `InprocServer32` y `LocalServer32` del registro de Windows para `HKEY_CLASSES_ROOT\CLSID\{619AF202-2174-11d2-9FFA-C68E0ECBEF02}` (figura 3.23).

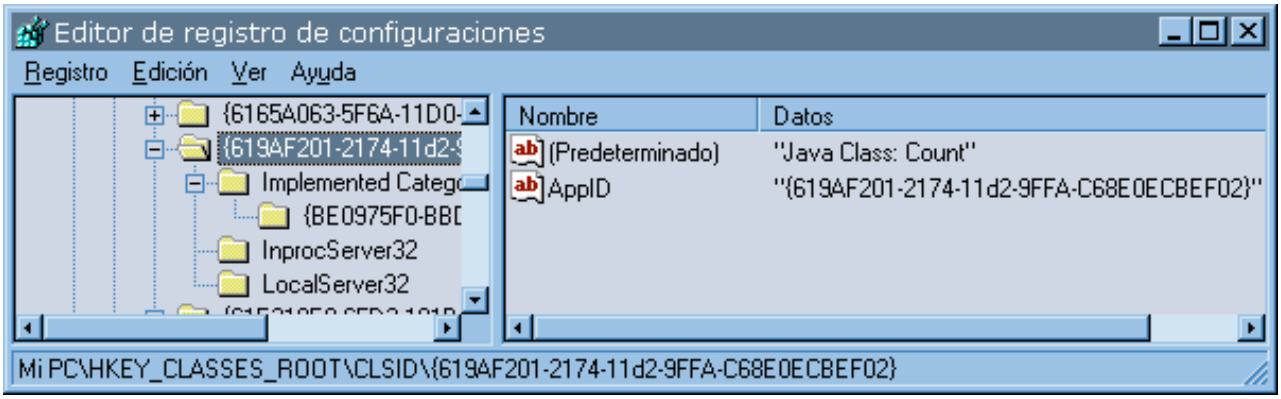

Figura 3.23: Edición del Registro de Windows.

A continuación se debe habilitar el servicio DCOM en ambas máquinas (donde residan cliente y servidor). Esto se hace a través de la utilidad de configuración DCOMCNFG (figura 3.24 en la página siguiente).

Seleccionando nuestra clase de entre las existentes en nuestro sistema (figura 3.25 en la página 92), podremos establecer sus propiedades (figura 3.26 en la página 93).

3.5.4 Ventajas e inconvenientes

A pesar de su engorrosa configuración, DCOM es un sistema de Objetos Distribuidos. Además, Microsoft tiene a esta tecnología en su punto de mira para los próximos Sistemas Operativos

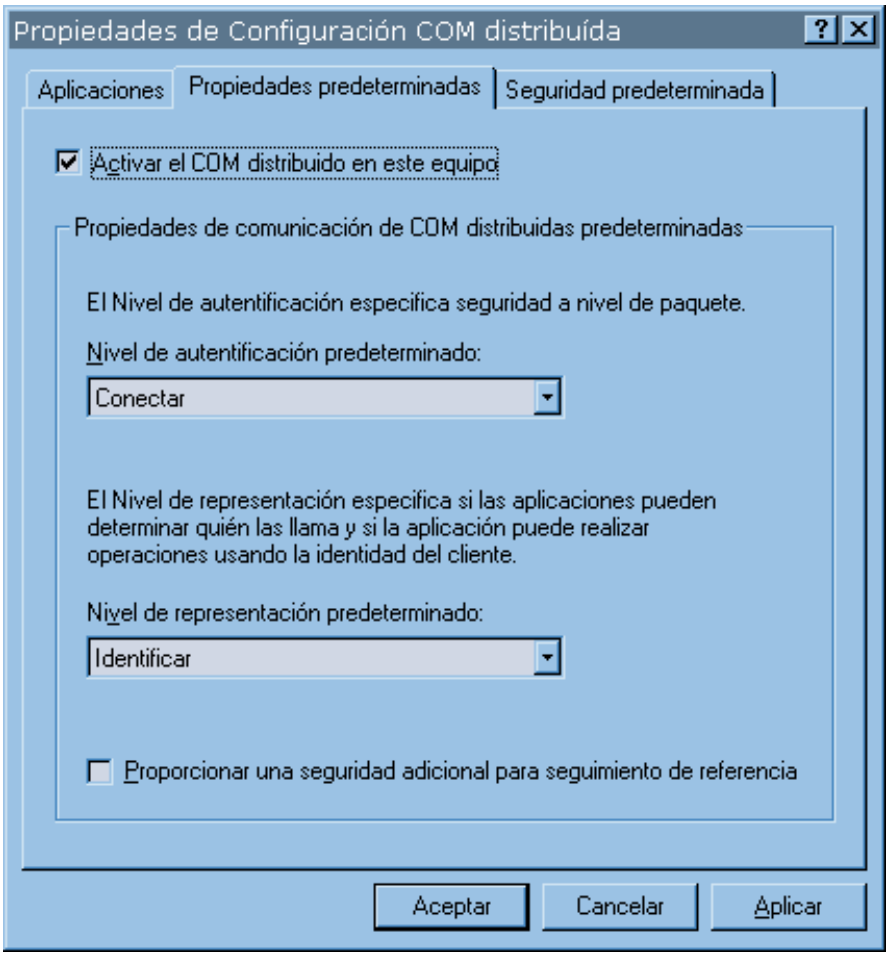

Figura 3.24: Habilitación de DCOM a través de DCOMCNFG.

(Windows 98 y NT 5.0). Esto implica que el desarrollo basado en DCOM quedará más integrado con las herramientas y con el Sistema Operativo muy pronto. Es por tanto una tecnología a tener muy en cuenta. De nuestro estudio de DCOM podemos obtener algunas **ventajas** al utilizar esta tecnología:

- ✔ **Posee un sistema de meta-información**. Existen dos puntos de información para un objeto DCOM: A través del Registro del Sistema, un componente puede saber los otros componentes instalados y conocer los CLSID de las clases que implementan; las librerías de tipos (usando el interfaz `ITypeLib`). Además, a través del método `IUnknown::QueryInterface()` se puede interrogar a cualquier objeto DCOM por los interfaces que implementa. Las ventajas de este mecanismo son importantísimas: un componente puede trabajar en conjunto y descubrir información sobre otros componentes que en el momento de diseñarlo no estaban previstos. Esto es particularmente importante, por ejemplo, para la reconfiguración automática de nuevos servicios que se añaden al sistema, o para herramientas RAD (*Rapid Application Development*, Desarrrollo Rápido de Aplicaciones) que son capaces de insertar nuevos componentes a aplicaciones (como por ejemplo, *Visual Basic* o *Borland Delphi*).

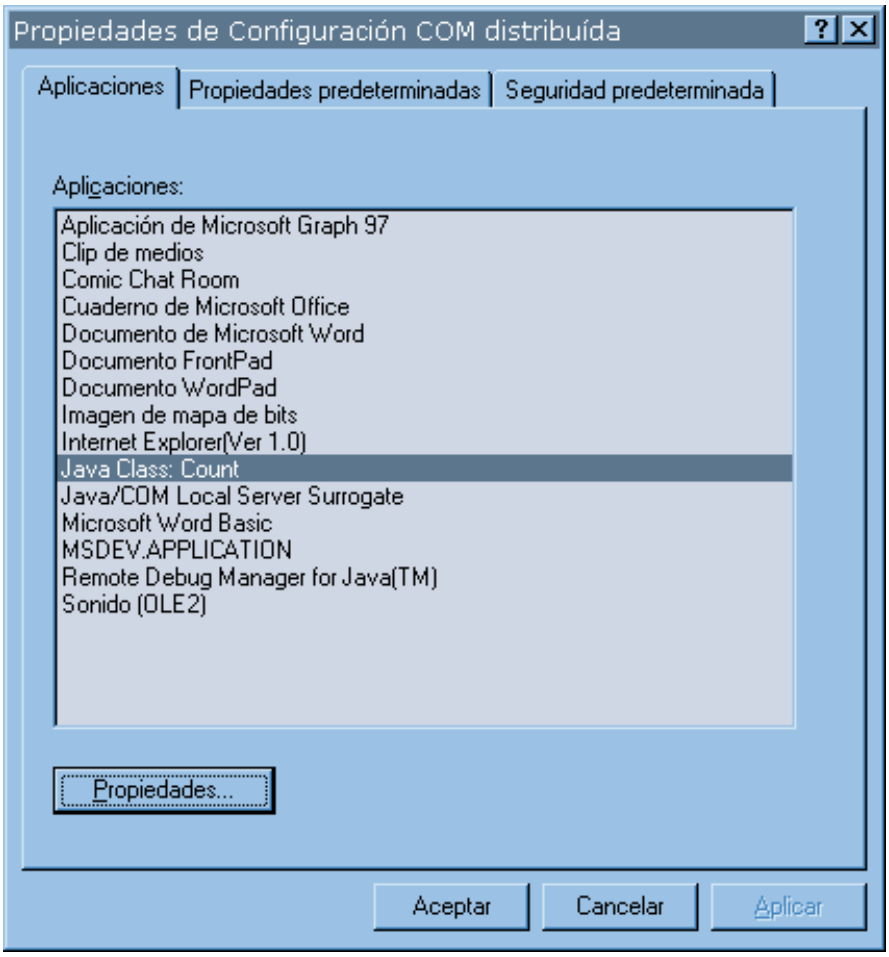

Figura 3.25: Clases DCOM configurables.

- ✔ **Permite invocaciones dinámicas**. En conjunto con el punto anterior, y a través del interfaz `IDispatch`, cualquier componente puede realizar llamadas construidas "*on the fly*" a nuevos componentes.

- ✔ **Logra (en teoría) transparencia local/remota**. En teoría porque hasta que no se mejore la configuración en la parte cliente, la transparencia local/remota queda algo deslucida. Sin embargo, una vez llevada a cabo esta configuración, las llamadas se realizan todas de la misma forma para el programador, independientemente de si el objeto servidor se ejecuta en esta u otra máquina.

- ✔ **Trabaja en torno a interfaces**. Como todos los sistemas de objetos distribuidos, se separa el interfaz de la implementación, y todos los componentes se construyen en base a las interfaces de los restantes, diseñadas en conjunto por el grupo de diseño. Esto da a las aplicaciones distribuidas las mismas características de modularidad, extensibilidad, reutilización, etc. de que gozan las aplicaciones realizadas en un entorno uniprocesador/lenguaje de programación único.

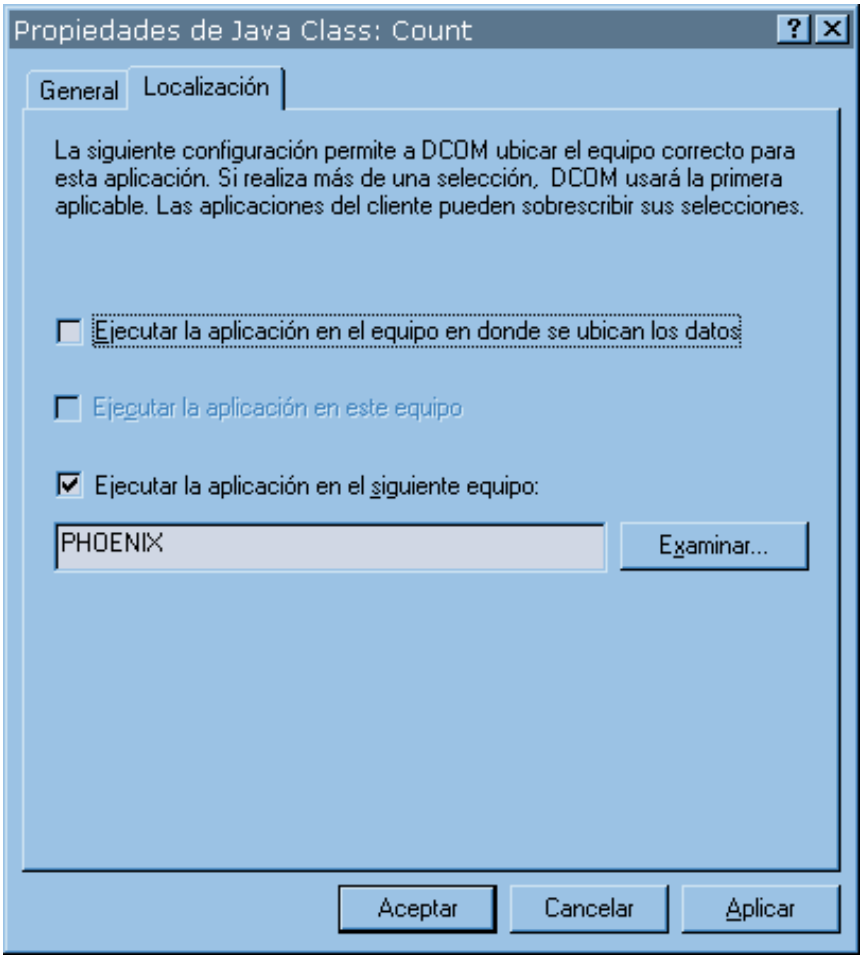

Figura 3.26: Configuración de la clase `Count`.

✔ **Es relativamente rápido**. Su velocidad es aceptable. Sin embargo, después veremos CORBA, nuestra apuesta para el desarrollo de aplicaciones con Objetos Distribuidos. CORBA es aún más rápido que DCOM. Aquí están los resultados de ejecutar el programa mostrado en el ejemplo con la misma configuración que en la sección 3.1.4

DCOM **Local** **(entre procesos)**	DCOM **Remoto** **(Ethernet a 10 Mbps.)**
¡No se pudo medir!	3.9 milisegundos

Tabla 3.12: Tiempo *ping* para DCOM.

(nótese que ni los autores de [OH97] pudieron ejecutar una simulación local. Desde luego, mucho tiene que cambiar DCOM para competir seriamente con CORBA, como veremos en el siguiente capítulo.)

✔ **Ofrece un conjunto de tipos estándar, así como el tratamiento de codificación y decodificación de los argumentos (*marshaling*)**. Esto hace que la tecnología sea

independiente del Sistema Operativo, de la plataforma hardware y de la red y el protocolo subyacente. Además, añade chequeo de tipos tanto estática como dinámicamente, a través de la meta-información dada por los interfaces y las librerías de tipos.

✔ **Se integra bien con las herramientas *Microsoft***. Esto lo convierte en ideal para ambientes en los que sólo se utilicen herramientas y entornos *Microsoft*, que por otro lado están teniendo, como ya hemos dicho a lo largo de este proyecto, un auge considerable.

Sin embargo, existen además una serie de **inconvenientes** que se harán más fehacietnes cuando estudiemos CORBA en el siguiente capítulo.

✘ **Su configuración es una auténtica pesadilla**. Como se vió en el ejemplo, la configuración del cliente es una auténtica desventaja de DCOM. Esto hace el proceso de lo que en Ingeniería del Software se ha dado en llamar "Gestión de la Configuración" (GC) realmente inabordable en un entorno donde nuevos componentes son añadidos al sistema, cambiados por nuevas versiones, cambiados de localización física, etc. En un sistema ideal, el cliente (es decir, el programa cliente que se ejecuta en la máquina del usuario) **no debe necesitar configuración**, o bien un único punto de acceso fijado desde el principio y que no cambie a lo largo del proceso de adaptación o cambio del sistema. En el siguiente capítulo veremos cómo CORBA ayuda a que esta GC no sea un problema.

✘ **No es escalable**. Esto es en parte por el punto anterior, y en parte porque el sistema de nombrado e identificación a nivel de LAN de *Windows* no es jerárquico. No se definen protocolos que conecten de forma estándar varias LAN y que permitan un sistema de Objetos Distribuidos "intergaláctico" ([OH97]).

✘ **No se basa en un modelo de objetos clásico**. Ya se vieron las consecuencias que esto introducía al principio de esta sección. La cruda realidad es que el modelo de objetos es más pobre, ya que, como se vió, no se soporta herencia múltiple de interfaces, sino sólo agregación. Además, los objetos no tienen un identificador único, lo que hace especialmente difícil el mantener un contexto de transacción en un entorno con fallos y desconexiones (como es Internet). Como veremos, CORBA ofrece un modelo de objetos clásico que soporta herencia múltiple polimorfismo, ligadura dinámica, invocaciones dinámicas, etc., haciendo más cómoda la programación al asemejarse a los lenguajes de programación Orientados a Objetos clásicos.

✘ **Sólo soporta herramientas específicas de *Microsoft***. Es sorprendente que esta característica sea a la vez una ventaja y un inconveniente. A nadie escapa la gran popularidad y ubicuidad de las herramientas *Microsoft*. Sin embargo, esto nos hace separarnos de estándares internacionales (como CORBA o Java) soportados por cientos de compañías de software. De cara a Internet, por ejemplo, sólo *Internet Explorer* soporta controles *ActiveX**. Además, sólo la máquina virtual Java de *Microsoft* (jview) es capaz de ejecutar clases DCOM.

*Que, por otro lado, han causado un gran revuelo, por su falta intrínseca de seguridad.

Capítulo 4

Aplicaciones Distribuidas Java/CORBA

Este capítulo presenta la tecnología CORBA, mostrando cómo Java y otros lenguajes de *script* con modelos de código móvil (*mobile code systems*) la han llevado a ser actualmente la mejor tecnología existente para desarrollo de aplicaciones Cliente/Servidor distribuidas.

CORBA, (*Common Object Request Broker Architecture*), esto es, "Arquitectura de Bus Común de Gestión de Peticiones de Objetos" ([OMG97], [VD98], [OMG98], [Ros97], [Cur97], [OH97], [OHE96], [MZ95]) es un *framework* estándar de Objetos Distribuidos creado por el consorcio OMG (*Object Management Group*), un consorcio de más de 760 empresas que se creó en 1989 con fundadores como *Hewlett-Packard* ó *Sun Microsystems*. Esta es una organización sin ánimo de lucro cuyos fines son promover la Orientación a Objetos en la ingeniería del software y el establecimiento de una plataforma arquitectural común para el desarrollo de programas basados en Objetos Distribuidos.

El primer paso en el camino del OMG fue la definición de la OMA (*Object Management Architecture*). Esta arquitectura representa el modelo en el que toda la aportación tecnológica del OMG es recogida. La OMA establece un marco en el que se incluye:

- Un Modelo de Objetos base (*Core Object Model*), que define los elementos básicos de la programación Orientada a Objetos (clases, objetos, implementación, interfaces, invocación, cliente, servidor, etc.).
- Un modelo de referencia, que establece el marco arquitectural. Éste identifica cinco componentes principales:
  - Un *Bus* común de gestión de peticiones y respuestas entre objetos en el que puedan ser integrados componentes de forma estándar. Este bus ha tomado el nombre de **ORB** (*Object Request Broker*), y se documenta en los estándares CORBA.
  - *Object Services* (Servicios de Objetos). Aquí se identifican un conjunto de servicios disponibles para los objetos. Éstos se espeficican en los documentos CORBAServices. Entre los servicios, se incluyen, el servicio de nombrado, eventos, ciclo de vida, transacciones, etc.
  - *Common Facilities* (Servicios Comunes). Se documentan en CORBAFacilities, y especifican servicios comunes a todos los objetos, como documentos compuestos, facilidades de Agentes Móviles, de Manejo de Sistemas, etc.

- Interfaces de Dominio. Son *frameworks* específicos para dominios de desarrollo, como por ejemplo, programas médicos, control del tráfico aéreo, etc. Todos ellos, al igual que los anteriores, especificados como interfaces.
- Interfaces de Aplicación. Estos son los interfaces que constituyen las aplicaciones específicas desarrolladas.

¿Pero cómo encaja CORBA en este marco? CORBA es un refinamiento del modelo de objetos base, es decir, extiende las definiciones y conceptos dadas por aquel, utilizando los mismos conceptos, pero más específicos y concretos. Establece, por ejemplo, la diferencia entre "Implementación de Objetos" y "Referencias de Objetos", la semántica de los interfaces, la semántica de las llamadas a operación, excepciones, etc. Además, como parte importantísima, define el lenguaje de especificación de interfaces: IDL* (*Interface Definition Languaje*), un lenguaje de especificación independiente del lenguaje de implementación utilizado. El documento de especificación de CORBA ([OMG97]) es bastante extenso en la definición de estos tópicos. A lo largo de este capítulo iremos tratando todos estos conceptos.

En la siguiente sección veremos qué ofrece Java a CORBA y qué ofrece este último al primero, lo que hace de esta una combinación perfecta para el desarrollo de aplicaciones distribuidas. En la sección 4.2 veremos una introducción a CORBA. Seguidamente (sección 4.3) veremos el lenguaje de definición de interfaces (IDL) y su *mapping* para Java. Después, en la sección 4.4) veremos la implementación de CORBA que hace VisiBroker de *Visigenic* y desarrollaremos una aplicación de ejemplo utilizando Java y CORBA. En la sección 4.5 veremos ciertas características avanzadas de CORBA, como son la interoperatividad, las invocaciones dinámicas, los *callbacks* y el uso de facilidades comunes estándar añadidas por el OMG a los servicios base del ORB. Por último, analizamos otro ORB escrito esta vez en Python, y demostramos así la capacidad del estándar CORBA de interconectar de forma transparente objetos implementados en distintos lenguajes de programación y en distintas plataformas.

4.1 Qué obtenemos con Java/CORBA

En esta sección justificaremos el por qué de la elección en conjunto de Java y CORBA. Esto se verá desde dos perspectivas: qué ofrece Java a CORBA y qué ofrece CORBA a Java. Esto mostrará, comparándolo con las secciones de ventajas e inconvenientes del capítulo anterior, cómo la unión de estas dos tecnologías sobrepasa con creces a las demás existentes. Como hasta ahora hemos hecho, también estudiaremos las ventajas que *otros* lenguajes de *script* ofrecen a CORBA, comparándolos con Java. Por último, veremos cómo Java y CORBA se integran con el WEB, ofreciendo una serie de ventajas que superan con mucho a las ofrecidas por la tecnología dominante en Internet: CGI. Comentarios en este sentido pueden encontrarse en [OH97, cap. 3], [VD98, cap. 1], [PWGB98, pág. xxii], etc.

4.1.1 ¿Qué ofrece Java™ a CORBA?

En anteriores secciones de este proyecto hemos visto características de Java. Su portabilidad y la buena integración con los *browsers* de Internet le hacen un lenguaje que tiene mucho que ofrecer a la programación distribuida y a CORBA. Sin embargo, aparte de su inmenso

*A partir de aquí, con IDL nos referimos al IDL de CORBA, que es diferente del (y no compatible con el) de DCOM.

plan de márketing y el respaldo de compañías como *Sun Microsystems*, las características más sobresalientes de Java son ahora compartidas por varios lenguajes de *script* de nueva generación, como Python o Perl.

A continuación se muestran una serie de ventajas que Java ofrece a la tecnología CORBA:

- **Portabilidad de las aplicaciones entre distintas Plataformas**. Una de las principales características de Java es que es a la vez compilado e interpretado. Esta característica no es nueva, ya que muchos lenguajes han hecho uso de ella (el primero de ellos, Pascal de Niklaus Wirth). La portabilidad entre plataformas se consigue dividiendo el proceso de compilación en dos fases: 1) la compilación hacia un código de una máquina virtual (*bytecode*); y 2) la interpretación, ya sea previa traducción al código máquina de la máquina destino (*Just In Time Compilation*) o no, de esos *bytecodes* teniendo como resultado la ejecución real. La ventaja es que el *bytecode* es independiente de la plataforma, y para cada una de ellas, se puede escribir un intérprete de *bytecodes* específico. El código en *bytecodes* puede ser pasado de una forma portable entre las distintas plataformas. Específicamente para CORBA, esto significa, por ejemplo, que el código de los servidores puede ser trasladado de un *host* a otro, dependiendo de las características cambiantes de la aplicación, no sólo sin cambiar los clientes al mantener intacto el interfaz (como permite CORBA *per se*), sino **sin tener que recompilar ni adaptar** el código al nuevo *host*, que posiblemente tendrá una plataforma Hardware+Sistema Operativo distinta. También significa que los clientes CORBA pueden viajar en forma de *applets* Java y ejecutarse en plataformas cliente distintas, con el único requisito de que soporten una *máquina virtual* Java (Java VM).

- **Programación para Internet**. Los clientes CORBA se pueden desarrollar como *applets* que se pueden integrar de forma directa en los *browsers* de la WEB. Esto proporciona a la tecnología CORBA un acceso inmediato al gran mundo de Internet.

- **Simplifica (o elimina) la Gestión de la Configuración en la parte del cliente**. Otra vez, utilizando *applets* como clientes CORBA, el coste de instalación de un nuevo cliente para una aplicación, o el coste de un cambio en el cliente tiene coste cero en términos de reconfiguración del cliente: símplemente el cliente debe descargar del servidor la nueva versión del *applet*. La gestión de la configuración e instalación se convierte en un proceso muy complicado para sistemas grandes Cliente/Servidor distribuidos. Con Java y CORBA estamos ante la tan deseada era del *Network Computer* (NC).

- **Un lenguaje de programación Orientado a Objetos relativamente sencillo y legible**. Java fue diseñado con varios objetivos en mente. Uno de ellos fue la simplicidad de la sintaxis del lenguaje, de alguna manera "trabajando sobre seguro". Tomando un lenguaje ampliamente aceptado, como es C++, y eliminando ciertas características engorrosas, como la herencia múltiple, la gestión de memoria o las clases parametrizadas (*template*), desarrollaron un lenguaje de sintaxis clara, orientado a objetos y con semántica de referencia.

- **Programación "multi-hilo" (*Multithreading*)**. Java soporta de forma integrada en su máquina virtual el uso de varios hilos de ejecución simultáneos, proporcionando monitores para conseguir exclusión mutua. Esto hace muy fácil el proceso de construir servidores que atienden a varios clientes de forma simultánea. El uso de *threads* es, como

vimos, una de las características que hacían a los *Servlets* sobresalir por encima de los programas CGIs tradicionales.

- **Simplifica la recolección de basura (*Garbage Collection*)**. Uno de los principales problemas en los sistemas distribuidos es la recolección de basura, esto es, la eliminación de los objetos a los que ningún cliente referencia. En C++, por ejemplo, el programador debe programar de forma específica cuándo y cómo sus objetos son destruidos. En Java esto es automático, y por lo tanto, la programación se hace más sencilla.

4.1.1.1 Comparación con otros lenguajes de *script*

Las ventajas que hemos señalado son muy válidas. Sin embargo, no nos debemos dejar llevar por "modas" ni por campañas de márketing en cuanto a lenguajes de programación. Actualmente hay otros lenguajes que comparten con Java muchas de sus características y que, por lo menos, están tan capacitados como éste. Citando a [Nic96],

> *No cabe duda de que la única razón por la que Java es popular es porque está diseñado y pensado para desarrollar programas que se ejecutan en el* browser. *Python se ha utilizado para construir un* browser *con características similares y puede ser considerado un competidor de Java si no fuera por la falta de* márketing.

es decir, la gran popularidad de Java es debida a su simbiosis con los *browsers* de la WEB. Sin embargo, otros lenguajes como Python o Perl que ofrecen características similares (también son orientados a objetos, compilan a *bytecode*, etc.) son serios competidores desde el punto de vista de la "adaptación" que estos lenguajes exhiben a la hora de escribir aplicaciones distribuidas.

Un punto importante en el que éstos fallan es en la cuestión de la seguridad. La mayoría de estos lenguajes permiten que el código del cliente se distribuya, bien en forma de *bytecodes* (como Java, Python), bien en forma de código fuente (Tcl, Perl). Esto requiere que se pueda garantizar una seguridad en el sentido de que cualquier trozo de código que se descargue en nuestro ordenador cliente no va a ocasionar la pérdida malintencionada de información. Perl y Tcl ofrecen "Safe-Perl" y "Safe-Tcl", pero el soporte que ofrecen para los browsers es muy limitado: en el caso de Tcl, existe un "Tcl Plug-in" que puede ser introducido en los *browsers* para que sean capaces de ejecutar código Tcl. Aún así, no consiguen la integración que provee Java.

Python, sin embargo, no ofrece ningún sistema de seguridad, aunque, como dice la cita, se ha desarrollado un *browser* escrito totalmente en Python con características similares a los que soportan Java (¡pero sin tanto márketing!). Este lenguaje ha sorprendido a varios autores por su simplicidad y la cantidad y calidad de su librería "built-in" (véase [Mul96], [Gar98] y la sección B.3). Por nuestra parte, analizamos Fnorb: un ORB CORBA 2.0 realizado íntegramente en Python en la sección 4.6.

4.1.2 ¿Qué ofrece CORBA a Java™?

En el capítulo anterior vimos dos tecnologías de Objetos Distribuidos, una nativa de Java: RMI, y otra adaptada para trabajar con Java como lenguaje de programación: DCOM. En esta sección veremos qué ofrece CORBA a Java que no ofrecían estas tecnologías. Estas

características junto con las secciones que siguen ayudarán a entender por qué la combinación de Java y CORBA da mejores resultados que las dos tecnologías mencionadas.

Lo que CORBA ofrece a Java puede resumirse en los siguientes puntos:

- **Interfaces definidos utilizando IDL**. La separación entre interfaz e implementación en CORBA está implícita, y se consigue definiendo todos los componentes de la aplicación e incluso los servicios genéricos disponibles utilizando un lenguaje de descripción independiente de la implementación: el IDL ó *Interface Definition Language*.

- **Un sistema de meta-información**. Gracias al *Interface Repository* (Repositorio de Interfaces), un objeto CORBA puede acceder a toda la (meta-)información sobre los interfaces que definen los demás objetos que están presentes en el sistema. Esto permite a los objetos localizar servicios genéricos o comunicarse y conocer otros objetos nuevos que se van integrando en el sistema dinámicamente. Esto hace muy sencillo el proceso de desarrollo y adaptación de las aplicaciones (debidos a cambios en la especificación, etc.) y convierte a todo el sistema en un gran repositorio para herramientas de programación RAD.

- **Permite que las peticiones se generen de forma dinámica**. Al poseer meta-información, los clientes pueden actuar de una manera que no se pensó en el momento de su creación, ejecutando métodos de forma dinámica en objetos que ni siquiera se conocían en el momento de la compilación del primero.

- **Independencia del lenguaje de programación utilizado**. Al contrario que RMI, por ejemplo, CORBA ofrece *ligaduras* (*language bindings*) con C, C++, COBOL, Smalltalk, Java, etc., que permiten que los métodos definidos en el IDL sean implementados utilizando cualquiera de estos lenguajes. Los clientes de estos objetos no son capaces (ni les interesa) discernir en qué lenguaje fueron implementados los objetos que les proveen servicio. Esta independencia no sólo ayuda a que cada parte del sistema sea implementada en el lenguaje adecuado para su funcionalidad, sino que permite la integración de viejos sistemas existentes en las empresas (*legacy systems*) dotándolos de un "nuevo look" e integrándolos de forma transparente en un sistema de Objetos Distribuidos estándar.

- **Transparencia de localización y activación del servidor**. El corazón de CORBA, el ORB ofrece a los objetos un mecanismo por el que pueden realizar invocaciones sobre objetos remotos de forma que éstas aparezcan ante el programador como locales. El ORB se encarga de alcanzar los objetos servidor reales y enrutar la petición en beneficio del usuario. Esto desacopla de la aplicación todo el manejo de la comunicación, dando la visión al programador de que trabaja con un "sistema grande", independientemente de si éste está compuesto de cientos de máquinas conectadas por redes heterogéneas.

- **Generación automática de Código *Stub* y *Skeleton***. Los sistemas distribuidos requieren mucha programación de bajo nivel (del estilo de la que vimos en la sección 3.1) para manejar el inicio, flujo y finalización de la comunicación, codificar y decodificar argumentos en el formato en el que se transmitirán, etc.; estos son los *stubs* del cliente y *skeletons* del servidor. A través de los compiladores de IDL, el código que realiza todas estas funciones se crea automáticamente. Un cliente de un objeto sólo necesita su IDL para construir de forma automática el código que le permitirá realizar invocaciones

remotas de forma transparente (más sobre esto en la sección 4.2). Recuérdese que estos dos puntos anteriores también los conseguíamos con RMI.

- **Reuso de** CORBASERVICES **y** CORBAFACILITIES. Los objetos que forman una aplicación distribuida normalmente requieren una serie de servicios adicionales. Estos servicios, en CORBA forman parte de lo que se conoce con CORBASERVICES y CORBAFACILITIES. Los servicios disponiblen incluyen un servicio de localización de objetos por nombre (esto es, obtener una referencia a un objeto conociendo su nombre único); servicio de localización de objetos por "tipo", es decir, obtención de referencias a objetos que "cubran nuestras necesidades", de forma similar a como se busca en las páginas amarillas; servicio de eventos, en el que los objetos se pueden registrar para ser notificados de ciertos eventos de interés; servicio de transacciones, de seguridad, etc. Estos servicios están ahí disponibles para ser reutilizados una y otra vez en las aplicaciones, liberando así a los desarrolladores de la carga de tener que implementarlos (véase sección 4.5.4).

- **Independencia del fabricante a través de la interoperatividad de los ORBs y la portabilidad de código**. CORBA2.0 define un protocolo estándar a través del que varios ORBs de distintos fabricantes se pueden comunicar de forma estándar (GIOP, *General Inter-ORB Protocol*). Esto significa que cualquier ORB que sea compatible con CORBA 2.0 puede ser accedido desde cualquier otro. El desarrollador puede utilizar el ORB del fabricante que mejores prestaciones de para una plataforma o lenguaje específico, sabiendo que, hacia el exterior, sus objetos van a presentar un interfaz uniforme y compatible. Esto deja el camino abierto a la competencia, pero "en la buena dirección".

- **Por supuesto, una velocidad muy superior a** CGI. Es claro que en el entorno Internet, CORBA debe competir con CGI. Por si todas las razones expuestas hasta ahora no han convencido de que CORBA es mucho más adecuado para el desarrollo de aplicaciones distribuidas que CGI, los siguientes datos de tiempo medio de invocación deben dejarlo claro si los comparamos con los obtenidos para CGI en la sección 3.2.5 (se usa la misma configuración):

CORBA **Local** **(entre procesos)**	CORBA **Remoto** **(Ethernet a 10 Mbps.)**
3.4 milisegundos	3.2 milisegundos

Tabla 4.1: Tiempo *ping* para CORBA.

lo cual, si observamos los datos de otras tecnologías, ésta es muchísimo más rápida que CGI (casi un 200%), y sólo es superada por los *Sockets* a bajo nivel. En la siguiente sección veremos cómo la unión de CORBA y Java se integran con el WEB.

En definitiva, CORBA nos permite programar para conseguir *funcionalidad*, independientemente de en qué lenguaje, *host* o plataforma hardware esté implementada: siempre está "lista para usar".

4.1.3 Java, CORBA y el Web

Antes de conseguir todas las ventajas que hemos visto en las dos anteriores secciones debemos tener en cuenta algunas consideraciones. En primer lugar, es cierto que cualquier *aplicación*

CORBA escrita en Java (nótese que utilizamos la palabra "aplicación", después abordaremos los *applets*) se puede transmitir hacia una máquina cliente y ejecutarse allí, siempre que ésta tenga un ORB que de soporte a una funcionalidad CORBA de la aplicación. Esta transmisión en forma de *bytecodes* de la aplicación puede hacerse incluso utilizando *Sockets* seguros (SSL, *Secure Sockets Layer*). Esta es una buena solución para un entorno Intranet en el que se tiene la seguridad de que las aplicaciones descargadas en forma de *bytecodes* poseen sólo código seguro en el entorno en el que se usan (el interior de una empresa, etc.)

Sin embargo, en todos los demas casos, no se puede tener certeza de que las aplicaciones CORBA escritas en Java que se ejecutarán en la máquina cliente no son malintencionadas. Por lo tanto, se requiere una seguridad similar a la garantizada por los *applets*: prohibición de escritura en el sistema de ficheros local, prohibición de uso directo de dispositivos de la máquina cliente, imposibilidad de conexión por red a una máquina distinta de la que se descargó el *applet*, etc. El uso de *applets* ofrece además la ventaja de integrarse perfectamente con los *browsers*, y por ello, con la WEB y convertir así al *browser* en el "cliente universal" (una vez más, el objetivo de los *Network Computer*). Nótese, sin embargo, que los *applets* se ejecutarán en el ambiente proporcionado por un *browser*, con lo que el propio *browser* no sólo debe ser compatible con Java y ser capaz de ejecutar *bytecodes*, sino que debe ser compatible con CORBA 2.0 y contener un ORB.

Los ORBs Java van todavía más allá eliminando la necesidad de que el *browser* contenga un ORB: implementan el ORB completamente en Java. Así, un *browser* que sea compatible con Java, a la vez que realiza la petición del *applet* puede realizar la petición de las restantes clases necesarias, que en este caso **implementan completamente** un ORB. Así, el único requisito expuesto ahora al cliente es que posea un *browser* compatible con Java. Un ORB implementado en Java puede ocupar unos 100 Kilobytes, y sólo se descargan las clases necesarias, lo que hace que el proceso no sea muy costoso.

De una manera abstracta, la secuencia de acciones que se llevan a cabo para ejecutar un *applet* CORBA es parecido a la interacción CGI, y es el siguiente:

1. **El *browser* pide la página HTML**. Utilizando HTTP, y de forma normal, el *browser* obtiene del servidor la página HTML que contiene al *applet* CORBA.

2. **El *browser* pide el *applet* que va embebido en la página**. Utilizando también HTTP, el *browser* descargar del servidor los *bytecodes* del *applet*.

3. **El *applet* se inicializa**, requiriendo para ello todos las clases *stub* que utiliza y el ORB implementado en Java para su funcionamiento CORBA. Las clases que implementan el ORB son entonces requeridas por el *browser* utilizando también HTTP y son instaladas en la máquina virtual Java del entorno del *browser*, donde también se está ejecutando el *applet*. El ORB queda entonces disponible para ser utilizado por el *applet*.

4. **El *applet* realiza sus peticiones a servidores** a través del ORB local y éste, a su vez, a través de IIOP, hacia los ORBs donde realmente se estén ejecutando los objetos implementación servidores.

El proceso se esquematiza en la figura 4.1 en la página siguiente (nótese que el ORB del servidor no tiene por qué ser un ORB Java; de hecho, no tiene que ser ni del mismo fabricante, símplemente ser compatible a nivel IIOP).

Este proceso todavía debe ser refinado. Si nos fijamos un poco más, vemos que las restricciones que se les imponen a los *applets* no permiten que éstos se conecten a ORBs de otras

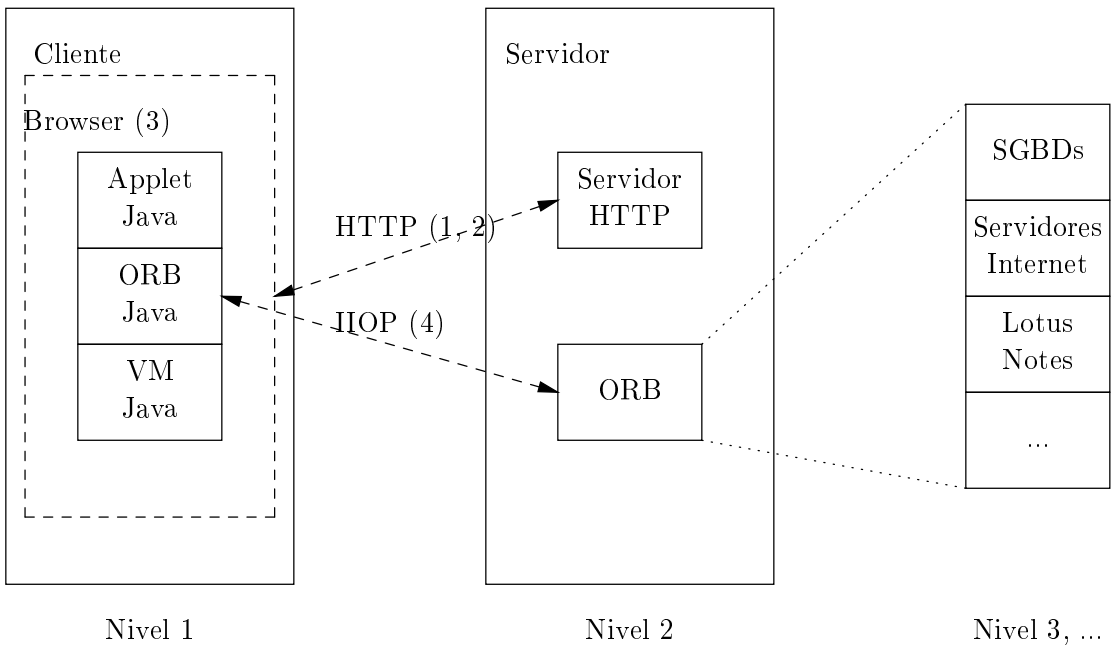

Figura 4.1: Arquitectura con *applets* CORBA.

máquinas que no sean la misma desde donde el *applet* se descargó. Además, la comunicación a través de IIOP no es posible si en la parte del cliente o del servidor existen *firewalls* (cortafuegos), que sólo permiten el puerto 80 (HTTP) y, posiblemente el 21 (FTP). La solución tanto a las restricciones de los *applets* como a las impuestas por los *firewalls* es construir lo que se llaman "pasarelas IIOP", que encapsulan los mensajes IIOP sobre HTTP. De este modo, aquellos se pueden transmitir por el mismo camino que las peticiones normales del *browser*. Al otro lado del cliente, debe existir un servidor que:

- Comprenda las peticiones HTTP y las sirva de forma normal.
- Tenga acceso al conjunto de clases Java que componen el ORB para enviarlo al cliente cuando éste lo solicite. Además, debe tener acceso a los distintos *stubs* (en forma también de clases compiladas a *bytecodes* Java) que el cliente necesitará.
- Procese de manera especial los mensajes IIOP embebidos en HTTP y realice la petición y recoja la respuesta en nombre del cliente real, enviándosela a éste de vuelta a través de la pasarela IIOP. Así el *applet* sólo se tiene que conectar a la máquina de la que vino.

La arquitectura queda definitivamente como se muestra en la figura 4.2 en la página siguiente.

La especificación CORBA no establece cómo se tienen que construir estos puentes. Cada producto que ofrece esta funcionalidad ofrece una herramienta específica que implementa la pasarela. En el caso de VisiBroker, es el programa gatekeeper. OrbixWeb de *IONA Technologies* ofrece un programa parecido. Los servidores de *Netscape Corp.* (*Enterprise Server*, etc.) ya lo llevan encapsulado, y el uso del gatekeeper no es necesario.

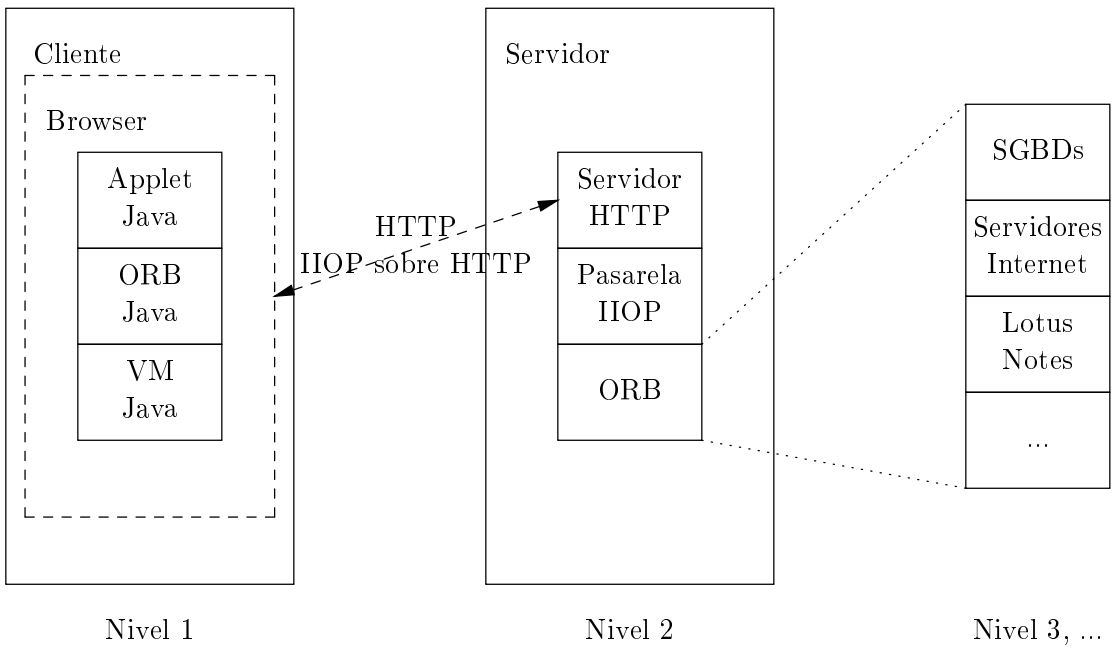

Figura 4.2: El uso de una pasarela IIOP sobre HTTP.

4.2 Introducción a CORBA

CORBA es una arquitectura estándar para el desarrollo de aplicaciones distribuidas basadas en Objetos. Permite que las clases que forman parte de las aplicaciones puedan ser implementados en distintos lenguajes, se ejecuten en distintas plataformas hardware+Sistema Operativo o estén dispersas por una red heterogénea.

Para conseguir esto, CORBA se centra en tres ideas clave:

- **La separación entre interfaz e implementación**. A través del lenguaje de definición de interfaces (IDL) se especifican todos los componentes CORBA. IDL es un lenguaje puramente declarativo con una sintaxis muy parecida a la de C++, pero sin estructuras programáticas. Es independiente del lenguaje utilizado en la implementación, existiendo ligaduras (*bindigs* ó *mappings*) para diversos lenguajes de programación (C, C++, Java, Ada, Smalltalk, COBOL, etc). Permite especificar las clases de las que un componente hereda, la signatura de operaciones, los atributos, las excepciones que lanza, y la signatura de métodos (incluyendo argumentos de entrada, de salida y valores de retorno y sus tipos de datos), etc. El *mapping* de IDL para Java se verá en la sección 4.3.

- **La independencia de localización**. El núcleo y componente más importante de cualquier implementación CORBA es el ORB (*Object Request Broker*). Éste se encarga de hacer transparente la localización de los objetos, enrutando las peticiones de manera que un objeto pueda comunicarse con otros independientemente de si ambos objetos se ejecutan en la misma máquina o en otras a través de redes heterogéneas. La estructura y funciones del ORB lo veremos en la siguiente sección (4.2.1).

- **La independencia del fabricante y la integración de sistemas a través de la interoperatividad**. A partir de la versión 2.0 de CORBA, se ha definido un estándar

para que ORB's de distintos fabricantes puedan integrarse en organizaciones heterogeneas y escalables de Objetos Distribuidos. El protocolo GIOP (*General Inter-ORB Protocol*), y su específico para Internet, IIOP, permiten a ORB's de distintos fabricantes comunicarse de una manera estándar. Esto, de cara al programador ofrece dos beneficios: 1) la independencia del vendedor; y 2) una invocación de métodos independiente de si ambos objetos (cliente y servidor) están en el mismo o en distintos ORB's. La interoperatividad la veremos en la sección 4.5.1.

4.2.1 El *Object Request Broker*, *ORB*

El ORB es el responsable de permitir a los objetos realizar de forma transparente las invocaciones y recibir respuestas de otros objetos en un ambiente distribuido, independientemente de si éste es un conjunto de nodos de procesamiento y redes homogéneo o heterogéneo.

En el ambiente CORBA, para que un objeto cliente pueda invocar operaciones en un objeto servidor (u "objeto implementación" en terminología CORBA), el objeto cliente tiene que obtener una *referencia* al objeto servidor. Como vimos en RMI, el proceso de invocación se divide en dos: la obtención de la referencia al objeto remoto y la invocación de la operación propiamente dicha. Una vez que un objeto ha obtenido una referencia a un objeto servidor (después veremos las distintas maneras que tiene un objeto de conseguir esto), puede realizar la invocación de los métodos y acceder a los atributos de este objeto, definidos a su vez a través del IDL de la clase a la que este último pertenece.

La figura 4.3 en la página siguiente muestra la estructura del ORB de CORBA. Ésta nos servirá para localizar todos los componentes y establecer la serie de servicios que están disponibles para los objetos CORBA.

Como dijimos, todos los servicios disponibles para los objetos CORBA están definidos utilizando IDL. En la figura se muestran los interfaces que el ORB ofrece a todos los objetos CORBA. Como se ve, el propio ORB ofrece un conjunto de servicios comunes (bajo el nombre de "ORB Interface") que pueden ser utilizados tanto por objetos clientes como por implementaciones. Entre las operaciones ofrecidas por el interfaz del ORB se encuentran la posibilidad de convertir referencias a objetos en cadenas de caracteres y viceversa para poder comunicar de manera sencilla referencias a objetos, obtener la clase de un objeto, etc.

Como en RMI y DCOM, tanto los clientes como los servidores necesitan unos "adaptadores" que transformen, en la parte cliente, una invocación local a una petición al ORB; y, en la parte servidor, una invocación por parte del ORB en una invocación en el objeto implementación (o servidor) real. En la parte cliente, estos trozos de código son denominados "Stubs". En la parte servidor, "Skeletons". CORBA también provee de interfaces para construir invocaciones de forma dinámica. Tanto en la parte cliente ("Dynamic Invocation Interface") como en la parte del objeto implementación ("Dynamic Skeleton Interface") se provee un interfaz para que los objetos puedan invocar (y recibir) llamadas a operaciones para las que no poseen *stubs*, especificando el objeto, el nombre de la operación que se invocará sobre él y los parámetros de la llamada. Estas facilidades dinámicas se apoyan en la meta-información dada por el "Interface Repository". Finalmente, el adaptador de objetos (*Object Adapter*) se apoya en el "Implementation Repository" para controlar el registro de servidores, activación y desactivación de implementaciones, instanciación en base a varias políticas, etc.

A continuación veremos con algo más de detalle todos estos componentes. Esto nos servirá para tomar una idea general de los distintos servicios que nos ofrece CORBA y nos preparará para, en la siguiente sección, estudiar el lenguaje de definición de interfaces y su

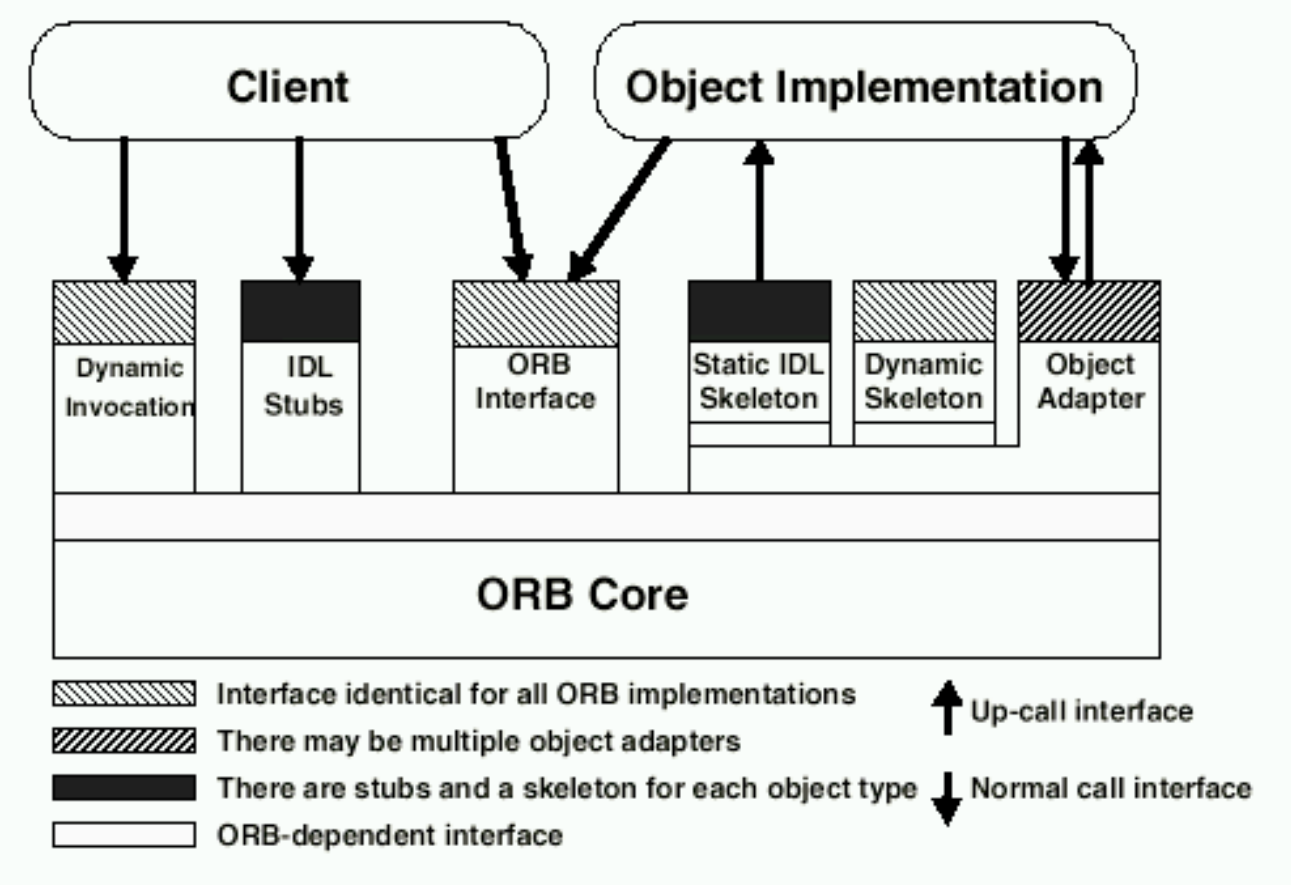

Figura 4.3: Estructura del ORB de CORBA (tomado de [OMG97]).

mapping para Java y posteriormente, en la sección 4.4 desarrollar nuestro primer programa basado en Java y CORBA.

4.2.1.1 El interfaz del ORB

Con los interfaces y servicios proporcionados por los *stubs* y los mecanismos de invocación dinámica, etc., quedan pocos servicios que sean de utilidad para ambos, clientes y servidores. El interfaz del ORB ofrece estos servicios a los objetos CORBA. Con sus dos operaciones, `string_to_object()` y `object_to_string()`, nos permiten transmitir referencias a objetos en forma de cadenas de caracteres. También ofrece operaciones de manejo de rencias, como son `is_nil()` para comprobar que una referencia es no nula, `duplicate()` y `release()`, que crean y destruyen, respectivamente, copias de un objeto; las funciones de meta-información `get_interface()` para obtener el "tipo" de un objeto y la función `is_a()`, que informa de si un objeto "puede ser visto" como de un tipo específico (esto es, si pertenece a una clase que hereda de cierta otra, en terminología de programación Orientada a Objetos convencional).

El ORB también es el punto inicial en el que los objetos pueden conseguir las referencias que necesitan. La operación `list_initial_services()` puede ser invocada por cualquier objeto para obtener referencias a los objetos que implementan servicios adicionales del bus,

por ejemplo y como vimos, el servicio de nombrado, el de transacciones, el de *Trading*, el Repositorio de Interfaces, etc.

4.2.1.2 Los *stubs* del cliente

Como hemos visto en varias ocasiones, el primer paso para invocar operaciones sobre un objeto remoto como si fuera local es la obtención de una referencia. Una vez que esta se ha obtenido, el objeto puede invocar operaciones sobre esa referencia como si el objeto servidor fuera local. Sin embargo, éste no lo es. El cliente necesita ser linkado con el "Stub" para el interfaz definido en IDL que le permita que las invocaciones que éste realiza de forma local se transformen en peticiones al ORB de ejecución de un método sobre el objeto remoto. En el objeto cliente, existe una operación (y un atributo) equivalente para cada una de los definidos en el IDL de la clase a la que pertenece. Los *stubs* son también llamados "objetos proxy". Los *stubs* son generados por el compilador de IDL para el lenguaje y el ORB específico que esté utilizando el cliente.

4.2.1.3 El interfaz de invocación dinámica (*DII*)

Por la descripción anterior, cabe deducir que cualquier cliente debe ser linkado con todos los *stubs* de los objetos servidores de los que pide servicios. Esto es así si se quiere utilizar invocación estática. Sin embargo, CORBA ofrece otro mecanismo más dinámico: el DII. Con este interfaz, un objeto cliente puede invocar cualquier operación en objetos para los que no posee *stubs*. Una vez que ha obtenido la (meta-)información neccesaria sobre el objeto sobre el que quiere realizar la invocación (como por ejemplo, saber sus métodos, número de argumentos de cada uno y tipo, etc.) puede construir un objeto del tipo `Request` (*Petición*) que encapsula toda la información sobre la invocación de un método a un objeto y la respuesta que este devuelve (incluyendo las posibles excepciones que se lanzan durante la ejecución). Esto, junto con el sistema de meta-información, hace a CORBA ideal para entornos dinámicos en los que se incluyen en el sistema nuevos componentes en tiempo de ejecución y en los que los componentes se descubren unos a otros y son capaces de trabajar en organizaciones que no fueron pensadas en el momento de la creación de ninguno de ellos de forma independiente.

Hay que hacer notar que de cara al servidor, una llamada estática o una utilizando invocación dinámica aparecen totalmente iguales: el servidor no distingue si su cliente lo invoca desde un *stub* estático o utilizando el interfaz dinámico.

4.2.1.4 Los *skeletons* del servidor

Al igual que el cliente necesita sus *stubs*, el servidor necesita sus *skeletons*. Para cada interfaz que un servidor implementa necesita un *skeleton*. Éstos son el equivalente para el servidor. Su función es transformar las peticiones que el ORB realiza (que son a su vez producidas por invocaciones de clientes, ya sea estática o dinámicamente) en invocaciones sobre el objeto servidor o implementación real, además de retornar de vuelta el resultado de la invocación (figura 4.3 en la página anterior). Estos *skeletons* son también generados por el compilador de IDL.

4.2.1.5 El interfaz de *skeletons* dinámicos (*DSI*)

El *Dynamic Skeleton Interface* permite a los servidores responder a peticiones de invocación de forma dinámica[†]. Esto significa que el servidor puede responder a peticiones interpretando en tiempo de ejecución el método invocado, los argumentos y su tipo y dándole semántica dinámicamente. La información la obtiene del interfaz `ServerRequest`. Esta técnica es útil para construir *wrappers* mínimos de componentes *legacy*, que símplemente pasan las peticiones al componente encapsulado, obteniendo a cambio una integración en el ORB.

4.2.1.6 El adaptador de objetos (*OA*, *Object Adapter*)

El adaptador de objetos se encarga del manejo de objetos implementación. Así, se encarga de la instanciación, activación, desactivación de objetos implementación, el manejo de sus referencias, la conexión de una invocación con su correspondiente objeto servidor, etc. El estándar CORBA define dos adaptadores: el BOA (*Basic Object Adapter*), genérico; y el POA (*Portable Object Adapter*), una estandarización más rígida y portable, por tanto.

Como un ejemplo de la labor del BOA, existen cuatro políticas de activación de objetos:

Shared Server (Servidor compartido). Si el proceso servidor contiene varios objetos. El BOA activa el proceso la primera vez que una petición se realiza a alguno de los objetos que el proceso implementa.

Unshared Server (Servidor no compartido). Si el proceso servidor contiene sólo un objeto. Para cada objeto cliente nuevo que requiera de este objeto implementación, se crea un nuevo para servirlo.

Server-per-Method (Servidor por método) En esta política, el BOA activa un servidor para cada método invocado, que termina al finalizar la ejecución del método.

Persistent Server (Servidor persistente) Aquí, el servidor se ha iniciado por otro agente distinto al BOA. Lo único que hace éste es enviarle las peticiones de los clientes.

Existe, además, otro adaptador específico para Sistemas de Gestión de Bases de Datos Orientadas a Objetos (OODBMS), en el que, por ejemplo, por defecto todos los objetos son persistentes.

4.2.1.7 El Repositorio de Interfaces (*IR*)

El Repositorio de Interfaces es un servicio que ofrece objetos persistentes que representan la información de IDL de manera que ésta es accesible en tiempo de ejecución. Esta meta-información puede ser utilizada por los clientes para construir invocaciones dinámicas. Este repositorio también es un lugar común donde se puede guardar información adicional sobre los interfaces, como información de depuración, etc.

Esta base de datos es actualizada por el compilador de IDL, y ofrece al programador un conjunto de clases que describen la (meta-)información que ésta posee y que permiten obtener esta información de una manera jerárquica.

[†]Una vez más, no importa si el cliente generó esta petición de invocación de forma estática o dinámica: son equivalentes. "Dinámico" se refiere esta vez al comportamiento del servidor frente a una petición.

4.2.1.8 El Repositorio de Implementaciones

El Repositorio de Implementaciones contiene información que permite al ORB localizar implementaciones de objetos. Esta información suele ser la del control de políticas, la información de instalación, y también otras informaciones adicionales, como información de depuración, control administrativo, reserva de recursos, seguridad, etc.

4.2.1.9 Comunicación entre ORBs: El protocolo IIOP

Como dijimos, el ORB se encarga de enrutar las invocaciones de un objeto cliente a un objeto servidor de forma transparente independientemente de si ambos están en el mismo ORB o en distintos, posiblemente de distintos fabricantes. Para conseguir esto, se necesita un estándar de interoperatividad entre ORBs. Esto lo da el protocolo GIOP (*General Inter-ORB Protocol*) y su específico de internet, IIOP: definen el conjunto de mensajes y la codificación a nivel del cable que se utiliza para comunicar ORBs.

Esta interoperatividad obliga a que las referencias a objetos dentro de un ORB se conviertan en referencias universales al comunicarse con otros ORBs. Estas referencias universales son las que se comunican entre ellos; son llamadas IORs (*Interoperable Object References*, Referencias Interoperables de Objetos). Una IOR contiene toda la información que permite a un ORB dirigir las peticiones y las respuestas hacia un objeto que reside en otro ORB. Entre otras cosas, y en el entorno Internet, como vimos en la sección 3.1, debe contener la dirección del *host* en el que el objeto reside, además de la referencia local del objeto en el ORB donde reside.

Actualmente hay un problema a la hora de transmitir IORs: no hay un protocolo estándar. Una vez que un objeto cliente lo obtiene (de la manera que sea), el ORB ya es capaz de ("mágicamente") hacer llegar las peticiones. En la sección 4.5.1, dedicada a la interoperatividad, veremos varias soluciones que no son más que parches hasta la aparición de un estándar.

4.3 IDL y el *mapping* de IDL a Java™

Como dijimos, IDL ([PWGB98], [OMG97]) es un lenguaje de especificación de interfaces independiente del lenguaje de implementación cuya sintaxis es muy parecida a C++[‡]. Éste estandariza los tipos y la estructura modular de manera que todos los objetos que populan el ORB las exhiban de igual forma. El estándar CORBA define cómo las distintas construcciones IDL se transforman en signaturas de métodos (o funciones, si el lenguaje de implementación no es Orientado a Objetos) y en atributos (o variables). Aquí se presenta el lenguaje IDL y su adaptación (*mapping*) para Java. Esta adaptación contempla la equivalencia de tipos de datos, de estructuras de programación y un punto muy importante: la portabilidad y estandarización de los *stubs* y *skeletons*.

4.3.1 Equivalencia de tipos de datos

La tabla 4.2 muestra la equivalencia entre los tipos de datos IDL y los tipos de Java. Nótese que el IDL soporta números con signo y sin signo, lo cual deberá ser tratado con mucho cuidado por los programadores Java. En la tabla también se muestran las distintas excepciones que pueden ser lanzadas en caso de un error de conversión.

[‡]De hecho, no estaría mal echar un vistazo a la biblia del C++ ([SE94]).

Tipo IDL	**Tipo Java**	**Excepciones**
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	
octet	byte	
string	java.lang.String	CORBA::MARSHAL
		CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHAL
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

Tabla 4.2: Equivalencia de tipos entre IDL y Java.

Después se verán las correspondencias para otros tipos más complejos, como las estructuras, las secuencias, los arrays y el tipo CORBA `Any`.

4.3.2 Módulos

A partir de aquí comenzamos a estudiar el *mapping* para Java siguiendo la estructura de módulos de IDL. Así, la estructura más externa es el módulo. Los módulos pueden contener interfaces u otros módulos, y así. En general, para una definición en IDL como la siguiente:

```
// -*- IDL -*-
module Window {...};
```

el código Java generado es:

```
// -*- Java -*-
package Window;
...
```

Para los módulos compuestos como:

```
// -*- IDL -*-
module org {
    module omg {
        module CORBA {
            ....
        };
        ...
    };
```

```
    ....
};
```

se genera:

```
// -*- Java -*-
package org.omg.CORBA;
```

Nótese sin embargo, que para referencia dentro de IDL, el nombre del módulo es `::org::omg::CORBA`, esto es, en vez de utilizar puntos (".") como en Java, se utilizan dobles dos puntos ("::") como en C++ para resolver el ámbito cuando sea necesario.

4.3.3 Interfaces

Dentro de un módulo pueden haber interfaces, excepciones, estructuras, uniones, enums, secuencias, arrays, etc. El *mapping* para los interfaces es mas complejo. Recuérdese que gracias a los interfaces se podían generar de forma automática los *stubs* y *skeletons*. Así, se generan varias clases por cada interface. La declaración IDL:

```
// -*- IDL -*-
interface Frame {...};
```

genera las siguientes clases:

- `interface Frame` (`Frame.java`). Este es el interfaz Java público de `Frame`. Los clientes desean referencias a objetos que implementen este interfaz, al estilo de la programación Java normal.
- `abstract public class FrameHelper` (`FrameHelper.java`). Esta clase es una clase abstracta que implementa métodos estáticos para operar con objetos de tipo `Frame`. En general, para un interfaz "`<interfaz>`", las clases `Helper` contienen, entre otros, los siguientes métodos (todos ellos `public static`):
  - `void insert(org.omg.CORBA.Any any, <interfaz> t)`, para crear un objeto del tipo `Any` que contenga un objeto de tipo `<interfaz>`. El tipo `Any`, como se verá en la sección 4.3.8, puede contener a cualquier otro tipo IDL definido. También existe la función `extract()` que realiza la función inversa,
  - `<interfaz> read(org.omg.CORBA.portable.InputStream _input)`, que ofrece una utilidad de serialización (también existe el método `write()`),
  - `<interfaz> narrow(org.omg.CORBA.Object obj)`. Este es uno de los métodos más importantes, ya que permite trasladar a un objeto a alguna de sus subclases (lo que en Java se conoce como "*narrowing*" y en C++ como "*casting*"). Esto es muy útil, por ejemplo, para convertir referencias obtenidas a través de los métodos del ORB `string_to_object()`.
  - `TypeCode type()`. Retorna un identificador del tipo tal y como será encontrado en el *Interface Repository*. Esto es muy conveniente para indicar el tipo de los parámetros y el resultado de las invocaciones que se construyen de forma dinámica a través del DII (*Dynamic Invocation Interface*).

Todas las clases definidas en IDL **son acompañadas por una clase** `Helper`.

- `final public class FrameHolder` (`FrameHolder.java`). IDL, como después veremos, soporta el uso de parámetros `in` (entrada), `out` (salida) e `inout` (entrada/salida). Java sólo soporta parámetros de entrada. Para ello, se crean unas clases "`Holder`" que "guardan" un valor de cada tipo. Para un interfaz llamado "`<interfaz>`", la clase puede ser la siguiente:

```
// -*- Java -*-
final public class <interfaz>Holder implements
    org.omg.CORBA.portable.Streamable {

public <interfaz> value;
public <interfaz>Holder() {}
public <interfaz>Holder(<interfaz> initial)
    { value = initial; }
public void _read(org.omg.CORBA.portable.InputStream i) { ... }
public void _write(org.omg.CORBA.portable.OutputStream o) { ... }
public org.omg.CORBA.TypeCode _type() { ... }
}
```

  Como se ve, la semántica de los métodos es fácilmente deducible de su signatura. Es de señalar que se genera una clase "`Holder`" **para cada tipo que se pase en un parámetro** `out` **ó** `inout`.

- `public class _st_Frame` (`_st_Frame.java`). Esta clase es el *proxy* del cliente para la clase `Frame`. Contiene, como ya vimos, los métodos para realizar invocaciones remotas, codificación (*marshaling*) y decodificación de los argumentos y los valores de retorno, etc.

- `abstract public class _FrameImplBase` (`_FrameImplBase.java`). Esta es la clase *skeleton* que implementa el interfaz `Frame`. Los servidores que implementen este interfaz deben heredar de esta clase, escribiendo el código para los métodos del interfaz.

En la sección 4.4 veremos un ejemplo sencillo en el que todas estas clases se verán en funcionamiento.

4.3.4 Estructuras

El *mapping* para estructuras (`struct`), uniones (`union`) y tipos enumerados (`enum`) es muy similar. Básicamente se genera la clase con el mismo nombre que la estructura, además de las típicas `Helper` y `Holder`. Como un ejemplo, el siguiente IDL

```
// -*- IDL -*-
...
struct Font {
    string    family;
    string    series;
```

```
    string    shape;
    short     size;
};
...
```

generaría

```
// -*- Java -*-
final public class Font
{
    public java.lang.String family;
    public java.lang.String series;
    public java.lang.String shape;
    public short            size;

    public Font() { }
    public Font(java.lang.String family,
                java.lang.String series,
                java.lang.String shape,
                short            size)
    {
        this.family = family;
        this.series = series;
        this.shape = shape;
        this.size = size;
    }

    // ...
}
```

donde no se muestran las clases `Helper` ni `Holder`.

4.3.5 Secuencias y arrays

El IDL de CORBA ofrece dos tipos de colecciones ordenadas: secuencias y arrays. Una secuencia se *mapea* a un array unidimensional de Java. Un array puede ser multidimensional y su longitud queda fijada en tiempo de compilación. Hay dos tipos de secuencias: limitada e ilimitada. Un ejemplo se puede ver en el siguiente código IDL:

```
// -*- IDL -*-
typedef sequence<string> ilimitada;
typedef sequence<short,30> limitada;
```

Este fragmento IDL no genera ninguna definición, tan sólo las clases `Holder`. El programador es el encargado de definir en el interior de su programa las variables adecuadas:

```
// -*- Java -*-
public java.lang.String[] valor;    // ilimitada
public short[]            valor;    // limitada
```

Igual ocurre con los arrays.

4.3.6 Atributos y operaciones

Dentro de un interface, como es normal en la mayoría de los lenguajes Orientados a Objetos, se pueden definir atributos y métodos (operaciones en terminología CORBA).

Los atributos son llevados a Java como un par de operaciones sobrecargadas, una de acceso y otra de cambio del atributo. Los atributos se pueden definir como de sólo lectura (`readonly`), en cuyo caso sólo el método de acceso es dado:

```
// -*- IDL -*-
module GraphicComponent {
    interface GenericComponent {
        attribute GenericComponent parent;
        readonly attribute string id;

        // ...
    };

    interface Button: GenericComponent {
        // ...
    };
};
```

generaría

```
// -*- Java -*-
package GraphicComponent;

public interface GenericComponent extends org.omg.CORBA.Object {

    public GenericComponent parent();
    public void parent(GenericComponent _val);

    public string id();

    // ...
}

public interface Button extends GraphicComponent.GenericComponent {
    // ...
}
```

y las clases `Helper` y `Holder` si es necesaria. Como se ve, dos métodos para el atributo de lectura y escritura, y un método para el que sólo es de lectura.

La definición de métodos tiene la siguiente sintaxis:

```
[oneway] <tipo> <identificador> (<parámetros>) [<raises>] [<context>]
```

donde `oneway` especifica que la operación se llama sólo una vez de forma informativa y no se espera a su ejecución (normalmente utilizado en métodos de finalización o de *ping*), `<tipo>`

especifica el tipo del valor devuelto (o `void` si no devuelve nada), `<identificador>` es el nombre de la operación. Los `<parámetros>` son una lista separadas por comas de valores:

```
{in | out | inout} <tipo> <identificador>
```

donde `in` especifica un parámetro de entrada, `out` uno de salida e `inout` uno de entrada y salida. `<raises>` utiliza la palabra "`raises`" y una lista entre paréntesis de excepciones que la operación puede lanzar y `<context>` utiliza la palabra "`context`" seguida de una lista de identificadores encerrados entre paréntesis y separados por comas. Esto último se utiliza para establecer el contexto de la operación, es decir, para pasarle información adicional.

Para permitir que los tipos estándar de Java puedan participar en parámetros `out` ó `inout`, se han añadido unas clases en el paquete `org.omg.CORBA`: `ShortHolder`, `IntHolder`, etc. tan sencillas como:

```
// -*- Java -*-
package org.omg.CORBA;

final public class ShortHolder {
    public short value;
    public ShortHolder() {}
    public ShortHolder (short initial) {
        value = initial;
    }
}
```

Como un ejemplo de *mapping* para una operación, considérese el siguiente ejemplo:

```
// -*- IDL -*-
interface Ajustador {
    void ajusta(in short numDecimales, inout double epsilon);
};
```

que generaría en Java:

```
// -*- Java -*-
public interface Ajustador extends org.omg.CORBA.Object {
    void ajusta(short numDecimales, DoubleHolder epsilon)
}
```

El uso que se haría de esta operación `ajusta()` sería el siguiente:

```
// -*- Java -*-

// Crear un objeto que implemente ese interfaz. No se muestra
// el código que le hace conectar con la implementación de la
// operación. Esto lo veremos después en la sección
// dedicada al ejemplo...
Ajustador a = new Ajustador();
```

```
// Parámetros. Para el tipo 'inout' se debe usar un 'Holder'
DoubleHolder epsilon = new DoubleHolder(0.122343);
short numDecimales = 3;

// Invocar la operación de ajuste
a.ajusta(numDecimales,epsilon);

// El nuevo épsilon ajustado está en 'epsilon.value'
```

donde se ve que para los parámetros `inout` (y también para los `out`) se debe utilizar una clase `Holder`.

4.3.7 Excepciones

CORBA, al igual que Java, también da la posibilidad de definir excepciones por parte del usuario. Además, CORBA también posee unas excepciones del sistema. Todas las excepciones del usuario extienden a `org.omg.CORBA.UserException` (que, a su vez, extiende a `java.lang.Exception`). Las excepciones del sistema extienden a `org.omg.CORBA.SystemException` (que extiende a `java.lang.RuntimeException`). Como ejemplo, considérese el siguiente código IDL:

```
// -*- IDL -*-
module Example {
    exception ex1 { string reason; };
};
```

generaría

```
// -*- Java -*-
package Example;
final public class ex1 extends org.omg.CORBA.UserException {
    public String reason;
    public ex1() { ... }           // Constructor por defecto
    public ex1(String r) { ... }   // Constructor
}
```

Hay varias excepciones del sistema, que son adaptadas como clases Java en el paquete `org.omg.CORBA`. Entre ellas, `CORBA::MARSHAL` a la clase `org.omg.CORBA.MARSHAL`, `CORBA::BAD_OPERATION` a la clase `org.omg.CORBA.BAD_OPERATION`, etc.

4.3.8 El tipo Any

El tipo IDL `Any` es algo especial. Se adapta a Java utilizando la clase `org.omg.CORBA.Any`. Esta clase soporta conversiones desde o hacia cualquier tipo IDL predefinido. La clase posee un constructor por defecto que construye un objeto `Any` vacío. La clase ofrece métodos `insert_<tipo>()` y `extract_<tipo>()`, donde `<tipo>` es el tipo IDL en cuestión. Los

métodos "insert" retornan un `Any`, y los "extract" retornan un `<tipo>`. Los últimos son declarados como posibles lanzadores de la excepción `org.omg.CORBA.BAD_OPERATION`.

Por ejemplo, para crear un objeto de tipo `Any` que contenga un `short`, podemos escribir

```
// -*- Java -*-
Any x = new Any().insert_short( 6 );
// Y ahora una actualización
x.insert_short( 12 );
```

Para el tipo `Any` es fundamental el interfaz `TypeCode` que se adapta a Java como la clase `org.omg.CORBA.TypeCode`. Este tipo define, valga la redundancia, el tipo de los objetos que populan en ORB y que pueden encontrarse en el *Interface Repository*. Cada interfaz definido en IDL, puede devolver un objeto `TypeCode` que indentifica unívocamente del tipo que es. Un `TypeCode` contiene un "kind"[§] (un entero que identifica al tipo) y unos parámetros específicos de ese tipo. Los identificadores de tipo para los tipos estándar IDL se encuentran en constantes estáticas de la clase `org.omg.CORBA.TCKind` como `TCKind.tk_<tipo>`. Existen, además, unos objetos `TypeCode` que representan a cada tipo estándar: `TCKind.tc_boolean`, `TCKind.tc_char`, `TCKind.tc_TypeCode`, etc.

4.3.9 *Stubs* y *Skeletons* portables

Cuando el *mapping* a otros lenguajes hubiera terminado, en Java se necesita algo más. La razón es clara: las clases *stub* y *skeleton* viajan por la red en forma compilada (*bytecodes*) y son cargadas en ORBs que pueden ser de distintos fabricantes. Se debe definir unos *stubs* y *skeletons* estándar de manera que puedan ser cargados directamente en tiempo de ejecución y funcionar independientemente del fabricante del ORB. Nótese que hasta la aparición del *mapping* a Java, el código interno de los *stubs* y *skeletons* no era especificado por CORBA y eran, por tanto, propietarios.

La portabilidad se consigue a través del paquete `org.omg.CORBA.portable`. Todos los *stubs* portables extienden a la clase `org.omg.CORBA.potable.ObjectImpl`. Todos los *skeletons* heredan de `org.omg.CORBA.DynamicImplementation`.

Es realmente admirable cómo se ha conseguido esta portabilidad. Los *stubs* son sorprendentemente sencillos, utilizan un interfaz común en todos los ORBs compatibles con CORBA 2.0: el *DII* (*Dynamic Invocation Interface*, Interfaz de Invocación Dinámica). Las llamadas al objeto *stub* del cliente se traducen en llamadas al interfaz de invocación dinámica con el método específico. Así los *stubs* son totalmente portables. Normalmente, los compiladores de IDL poseen una opción para generar este tipo de *stubs*, ya que normalmente son más lentos que los propietarios, al utilizar invocación dinámica. En VisiBroker, la opción a idl2java es `-portable`. Esto genera una clase llamada `_portable_stub_<interfaz>`.

Para los *skeletons* se utiliza el interfaz simétrico: el *DSI* (*Dynamic Skeleton Interface*, Interfaz de `Skeletons` dinámico). La clase de la que heredan, `DynamicImplementation` tiene la siguiente signatura:

```
// -*- Java -*-
package org.omg.CORBA;
```

[§]Es difícil diferenciar en español las palabras inglesas "type" y "kind". En general son equivalentes, como en este caso, en el que "kind" es otra manera de llamar a un "type".

```
public abstract class DynamicImplementation
                        extends org.omg.CORBA.portable.ObjectImpl
{
    public abstract void invoke(org.omg.CORBA.ServerRequest request);
}
```

Implementando el método `invoke()`, los *skeletons* pueden recibir todo tipo de peticiones, ya que la información de cada una se guarda en el objeto `ServerRequest`. Así, tanto *stubs* como *skeletons* se construyen a partir del API estándar de CORBA y son, por ello, portables.

4.4 VisiBroker™ for Java™

Una vez establecida la teoría que rodea a CORBA, estamos en condiciones de realizar una aplicación de ejemplo que muestre cómo todos los elementos y clases que hemos introducido se coordinan para conseguir que todo funcione como deseamos.

Utilizaremos para esta primera aplicación un ORB Java que se ha hecho muy popular: VisiBroker for Java 3 de *Visigenic Software* (ahora *Inprise* tras la adquisición por *Borland Intl.*). Extensa bibliografía sobre este ambiente se puede encontrar en [VIS97b], [VIS97a], [PWGB98], [OH97], etc.

VisiBroker ofrece un entorno de programación con un compilador de IDL a Java (idl2java), un servidor HTTP con pasarela IIOP (llamado gatekeeper) para labores de desarrollo (es decir, para probar las aplicaciones) y un ORB escrito en Java. Como parte de las características específicas de este ORB se encuentran un servicio de nombrado limitado que permite encontrar referencias a objetos a través de un nombre en una misma red IP, un compilador llamado *Caffeine*, que permite obtener *stubs* y *skeletons* a partir de clases Java, etc.

Aunque esta última opción ofrece muchas posibilidades, no es todavía estándar, y no la estudiaremos aquí. En cuanto al sistema de nombrado propietario, es conveniente para la primera aplicación. Después, en la sección 4.5, veremos cómo distinguir entre servicios propietarios y servicios estándar CORBA. En particular, a obtener referencias a objetos de forma estándar a través de IORs que son inmediatamente utilizables por cualquier ORB CORBA 2.0. Como vimos, resulta muy conveniente para los desarrolladores de aplicaciones CORBA arroparse en el estándar para estar a salvo del encadenamiento a soluciones propietarias.

4.4.1 Un ejemplo simple paso a paso: un banco

La primera aplicación que veremos es una muy simple y del dominio en donde CORBA está teniendo un mayor éxito: la banca. El programa es una simple consulta del saldo de cuentas de un usuario, indexadas por nombre. En la parte cliente, un interfaz gráfico de usuario (GUI) permitirá escribir el nombre del usuario del que se quiere saber el saldo. Esto causará que se realice una invocación a un objeto CORBA implementado en Java para obtener el valor de la cuenta. La aplicación mostrará un ejemplo sencillo, aunque completo, de una interacción CORBA y de la integración de CORBA con el Web, ya que el cliente es un *applet* Java. A la vez, servirá de "esquema" del proceso de desarrollo usando la tecnología CORBA.

Comenzaremos, como siempre, por la definición del IDL para los interfaces que participan en la aplicación. Después tendremos que ejecutar el compilador de IDL para conseguir las clases Java base para la implementación tanto del cliente como del servidor. Una vez que

hayamos implementado la funcionalidad de la aplicación, habrá que ejecutarla con la ayuda de gatekeeper.

4.4.1.1 IDL

Esta es una aplicación sencilla, por lo que sólo consta de un módulo IDL en un fichero. La definición es la siguiente:

```
// Bank.idl

module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

Nótese que se definen dos interfaces: `Account`, que encapsula una cuenta y ofrece una operación para obtener el saldo (`balance()` de la misma; y `AccountManager`, que gestiona la creación de nuevas cuentas para usuarios, ofreciendo éste una operación, `open()`, que abre una cuenta a un nuevo cliente.

4.4.1.2 Generación de los ficheros

A continuación debemos ejecutar el compilador de IDL para obtener las clases que nos permiten implementar la funcionalidad de los interfaces definidos, los clientes y los servidores. Para ello, invocaremos a idl2java:

```
idl2java Bank.idl
```

que genera (entre otros) los siguientes ficheros en el directorio "`./Bank/`":

```
Account.java              AccountManager.java
AccountHelper.java        AccountManagerHelper.java
AccountHolder.java        AccountManagerHolder.java
_AccountImplBase.java     _AccountManagerImplBase.java
_example_Account.java     _example_AccountManager.java
_sk_Account.java          _sk_AccountManager.java
_st_Account.java          _st_AccountManager.java
```

Todos los ficheros parecen los mismos que dijimos en la sección 4.3.3. Sin embargo, hay dos por cada interfaz que no vimos: El fichero `_sk_Account.java` corresponde al *skeleton* del servidor de versiones anteriores de VisiBroker. Se da por compatibilidad hacia atrás; y el fichero `_example_Account.java`, que muestra un ejemplo de cómo debería ser la implementación para cada uno de los interfaces.

4.4.1.3 La aplicación

La aplicación debe ser ahora implementada. Esto significa que tenemos que implementar los métodos definidos en IDL para cada interfaz. También debemos escribir el servidor que lance las implementaciones servidoras de cada uno de estos interfaces y la aplicación cliente que utilice estos servicios para devolvernos el saldo de la cuenta en cuestión.

No hay nada más simple que la implementación de la funcionalidad del interfaz `Bank.Account`. Como dijimos, debe extender la clase `Bank._AccountImplBase` e implementar el interfaz `Bank.Account`. La clase se llamará `AccountImpl`, aunque no se requiere que tenga ningún nombre especial: su posición en la jerarquía de herencia determina qué interfaz implementa. Aquí está el listado:

```
// AccountImpl.java

public class AccountImpl extends Bank._AccountImplBase {
  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _balance;
  }
  private float _balance;
}
```

(la cláusula `implements` queda implícita por la `extends`, ya que la clase base implementa el interfaz `Bank.Account`). Nótese cómo la implementación sólo se tiene que preocupar de la **funcionalidad real del interfaz**. Toda la infraestructura de comunicaciones y de invocación remota es abstraída por el sistema.

Igual de simple es la implementación de la funcionalidad del interfaz `Bank.AccountManager`. Se utiliza la clase `AccountManagerImpl` para eso:

```
// AccountManagerImpl.java

import java.util.*;

public class AccountManagerImpl extends Bank._AccountManagerImplBase {
  public AccountManagerImpl(String name) {
    super(name);
  }
  public synchronized Bank.Account open(String name) {
    // Localizar la cuenta en el diccionario
    Bank.Account account = (Bank.Account) _accounts.get(name);
    // Si no existe en el diccionario, crear una
    if(account == null) {
      // Inicializar el saldo entre 0 y 1000 dólares.
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // Crear la implementaci'on de la cuenta, dado el saldo.
```

```
      account = new AccountImpl(balance);
      // Hacer al objeto visible por el ORB.
      _boa().obj_is_ready(account);
      // Imprimir la cuenta.
      System.out.println("Creada la cuenta para: "+name+": "+account);
      // Guardar la cuenta en el diccionario.
      _accounts.put(name, account);
    }
    // Retornar la cuenta.
    return account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
}
```

La funcionalidad es sencilla. Al invocar la función `open()`, si el nombre dado ya está en el diccionario `_accounts`, se retorna el objeto `Bank.Account` asociado. Nótese que se devuelven referencias a objetos locales. El ambiente de ejecución del ORB (concretamente el *Object Adapter*) debe convertir esas referencias locales a otras globales al ORB. Si el nombre dado no está asociado con ningún objeto `Account`, se debe crear uno con un balance escogido al azar. Esta creación se realiza con `new` sobre la clase que implementa al interfaz: `AccountImpl`. Finalmente, si el *Basic Object Adapter* es el que se encarga de manejar las referencias a objetos implementación, se le debe comunicar cada vez que se cree uno de estos objetos. Esto se hace con la llamada "`_boa().obj_is_ready()`".

Hay todavía un último detalle importantísimo a considerar: la definición del método posee la palabra clave `synchronized`. El usar esta palabra clave hace que el método se convierta en una sección crítica: dos o más objetos clientes que quieran acceder al método `open()` lo harán en exclusión mútua. Esto asegura que el objeto implementación es capaz de servir a varios clientes simultáneamente (en inglés, "*thread-safe*").

El servidor debe crear un objeto de la clase `AccountManagerImpl` que servirá a las peticiones de los clientes. El código se ejecuta en la función estática `main()`:

```
// Server.java

public class Server {

  public static void main(String[] args) {
    // Inicializar el ORB.
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    // Inicializar el BOA.
    org.omg.CORBA.BOA boa = orb.BOA_init();
    // Crear el objeto "AccountManager".
    Bank.AccountManager manager =
      new AccountManagerImpl("BankManager");
    // Exportar el objeto recién creado.
    boa.obj_is_ready(manager);
```

```
    System.out.println(manager + " está disponible.");
    // Esperar las peticiones
    boa.impl_is_ready();
  }
}
```

Las dos primeras líneas inicializan el ORB y el BOA (*Basic Object Adapter*) de cara a la aplicación. La siguiente línea crea un objeto del tipo `Bank.AccountManager` a través de la clase que implementa su funcionalidad. El argumento al constructor es un nombre, "`BankManager`", que permitirá que los clientes se puedan conectar con él de forma directa utilizando su nombre. Como comentamos, esta es una característica específica de VisiBroker. Se debe notificar al BOA que un nuevo objeto implementación se ha creado. Esto se hace con la invocación `boa.obj_is_ready()`, dándole como parámetro el objeto implementación creado. El servidor debe entonces quedar en un bucle esperando las distintas peticiones de los clientes. Esto se consigue con la llamada en la última línea: `boa.impl_is_ready()`.

Un servidor puede lanzar varios objetos implementación y se puede adherir a las distintas políticas de activación que provee el BOA (sección 4.2.1.6).

Es tiempo, finalmente, para el cliente. Éste será implementado como un *applet* Java. Por ello, este código resulta doblemente importante: es una solución real a los problemas que vimos en la sección 4.1.3 y además, es un cliente CORBA. Su código, que comentaremos a continuación, es el siguiente:

```
// ClientApplet.java

import java.awt.*;

public class ClientApplet extends java.applet.Applet {

  private TextField _nameField, _balanceField;
  private Button _checkBalance;
  private Bank.AccountManager _manager;

  public void init() {
    // Este GUI utiliza un enrejado de 2x2 elementos.
    setLayout(new GridLayout(2, 2, 5, 5));
    // Añadir los cuatro elementos.
    add(new Label("Nombre de la Cuenta"));
    add(_nameField = new TextField());
    add(_checkBalance = new Button("Comprobar Saldo"));
    add(_balanceField = new TextField());
    // Hacel el campo "balance" no editable.
    _balanceField.setEditable(false);
    // Inicializar el ORB (usando el Applet).
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(this, null);
    // Localizar un "AccountManager"
    _manager = Bank.AccountManagerHelper.bind(orb, "BankManager");
  }
```

```
  public boolean action(Event ev, Object arg) {
    if(ev.target == _checkBalance) {
      // Pedirle al AccountManager que abra una cuenta con un nombre.
      // Coger el nombre del campo "_nameField".
      Bank.Account account = _manager.open(_nameField.getText());
      // Actualizar el balance en el GUI.
      _balanceField.setText(Float.toString(account.balance()));
      return true;
    }
    return false;
  }
}
```

En el método `init()` se inicializan todos los componentes visuales del *applet*. También se inicializa el ORB y se liga a un objeto que implementa el interfaz `Bank.AccountManager` con el objeto implementación de ese interfaz que habíamos llamado "`BankManager`". Esto se hace en la última línea de este método, a través del método estático `bind()` de la clase `Bank.AccountManagerHelper`. Este método `bind()` no es estándar, y es específico de VisiBroker. Utiliza el sistema de nombrado propietario sencillo que vimos en el programa. Como dijimos, en la sección 4.5 veremos cómo utilizar clientes que no estén ligados a esta implementación propietaria. Sin embargo, esto es muy difícil de aplicar a clientes que son *applets*.

La inicialización del ORB[¶] a través de la función `init( this, null )` que acepta al *applet* como parámetro hace que el comportamiento del *stub* cambie completamente: como parte de su inicialización, establecerá una conexión con una instancia del gatekeeper de VisiBroker, del cual se espera que se esté ejecutando en la misma máquina desde la que se recuperó el *applet*. gatekeeper es a la vez un objeto CORBA y un servidor HTTP (véase sección 4.1.3).

Cuando el botón es pulsado, el sistema de eventos del AWT (*Abstract Windowing Toolkit*) ejecuta el segundo método: `action()`. Éste lee el valor en el campo de texto y lo utiliza como argumento al método `open()` del objeto `Bank.AccountManager` anteriormente creado. Este método retorna otro objeto de tipo `Bank.Account`, al que ahora se le puede hacer la petición `balance()` para obtener su saldo.

La simplicidad del mecanismo es sorprendente. No se necesita conocer ninguno de los detalles de la comunicación entre objetos remotos. Símplemente invocar a un método de forma normal. Las clases *stub* (y *skeletons* si son necesarias) se descargan gracias al *browser* cuando se necesitan, y éstas se encargan de implementar toda la semántica remota.

Por último, la página HTML que contiene al *applet*. Ésta contiene la definición del *tag* HTML que hace que el *browser* cargue e inicialice la clase `ClientApplet`:

```
<h1>VisiBroker for Java Client Applet</h1>
<hr>
<center>
  <applet
    code=ClientApplet.class
    width=200 height=80>
```

[¶]Nótese que no hay inicialización del BOA: para los clientes resulta inútil.

```
    <param name=org.omg.CORBA.ORBClass value=com.visigenic.vbroker.orb.ORB>
    <h2>You are probably not running a Java enabled browser.
    Please use a Java enabled browser (or enable your browser for Java)
    to view this applet...</h2>
  </applet>
</center>
<hr>
```

Si observamos, al *applet* se le pasa un parámetro, "`org.omg.CORBA.ORBClass`" con valor "`com.visigenic.vbroker.orb.ORB`". Esto indica qué clase ORB utilizar. Normalmente esto no debe ser especificado. Sin embargo, *Netscape Navigator 4.x* lleva integrado un conjunto de clases que implementan el ORB de VisiBroker, pero en su versión 2.5, que es algo diferente de la versión 3. Este parámetro hace que no se utilicen las clases que el navegador posee; en vez de eso, se descargan por la red.

4.4.1.4 Hacer que la aplicación funcione

Una vez que hemos escrito todos los componentes de la aplicación debemos compilarla utilizando el compilador de VisiBroker: vbjc. El uso de este compilador no es obligatorio. De hecho, éste utiliza en última instancia el compilador del JDK que tengamos instalado. El único requisito para utilizar javac es añadir al `classpath` las clases que implementan el ORB (bajo el directorio `/vbroker/lib`). Debemos compilar el cliente y el servidor (como es normal, el compilador de Java compila todas las clases que éstos utilizan, en particular las clases *stub*, etc. en el directorio `./Bank/`). La orden es la siguiente:

```
vbjc ClientApplet.java
vbjc Server.java
```

Antes de iniciar el servidor en la máquina elegida, como hemos utilizado el sistema de directorio de VisiBroker, debemos utilizar el programa que actúa de *demonio* (*daemon*) traductor de nombres a referencias de objetos implementación. Este demonio se llama *VisiBroker Smart Agent*. En una misma LAN, no importa qué ordenador ejecute este demonio: utilizan *Sockets multicast*. En Windows 95/NT, el *Smart Agent* se ejecuta haciendo doble *click* en el icono (figura 4.4).

Figura 4.4: *Smart Agent* de VisiBroker.

A continuación, debemos lanzar el servidor. Para ello se utiliza el programa vbj. Otra vez, este programa sólo llama al intérprete Java (java), pero con un `classpath` tal que incluye al ORB Java de VisiBroker. El servidor se lanza con

```
vbj Server
```

Para que el *applet* funcione también debemos lanzar el gatekeeper. Éste se debe lanzar en el mismo directorio donde se encuentra el *applet* y la página HTML. Es claro que esta solución es provisional e indica que esta herramienta es válida sólo en el proceso de desarrollo. La línea de comandos sería:

```
start gatekeeper
```

o "`gatekeeper &`" si estamos bajo algún UNIX. Como servidor HTTP, gatekeeper se instala en el puerto 15000, por lo que la dirección de la página HTML de nuestra aplicación sería "`http://<host>:15000/ClientApplet.html`", donde "`<host>`" indica la máquina en donde se ejecuta gatekeeper. Apuntando a nuestros *browsers* a esta dirección obtenemos las siguientes páginas (figuras 4.5 y 4.6 en la página siguiente).

Figura 4.5: *Applet* CORBA en *Internet Explorer 4*.

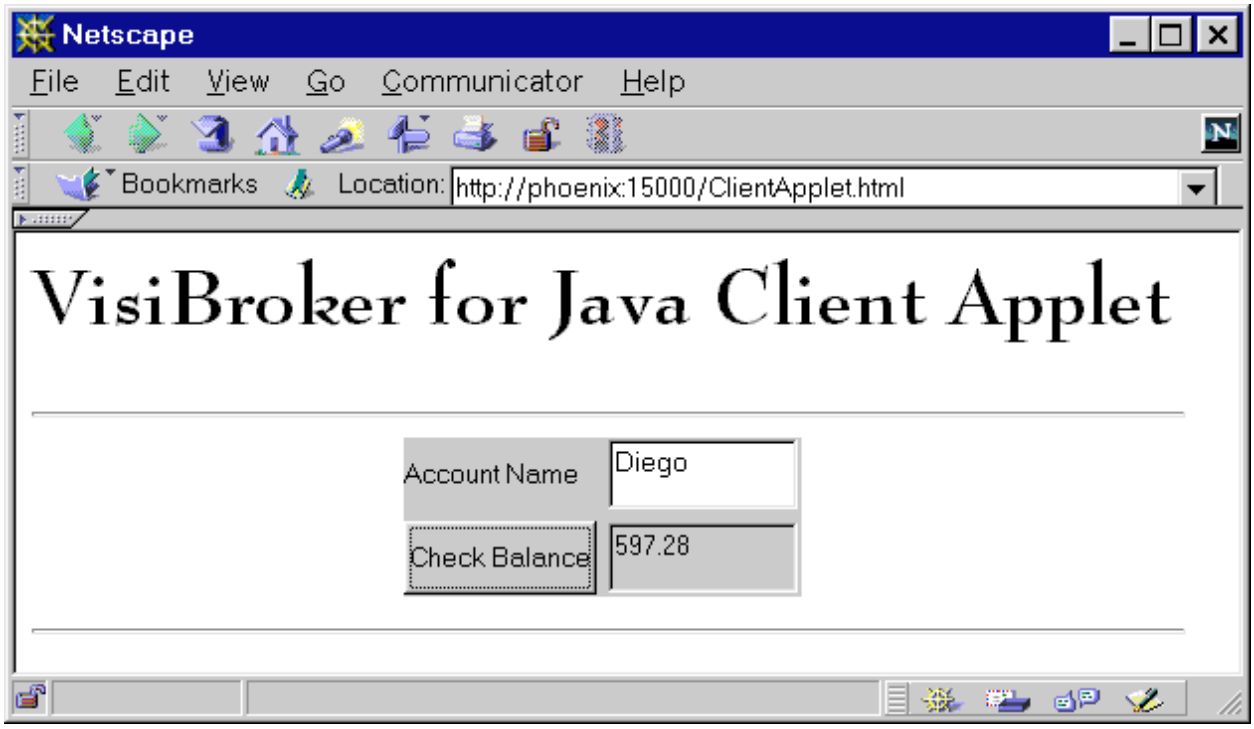

Figura 4.6: *Applet* CORBA en *Netscape Navigator 4*.

4.4.1.5 Profundizando un poco más

El hecho de que `gatekeeper` sea un servidor HTTP y provea unos *logs* nos hace que podamos analizar un poco más el proceso de ejecución del *applet*, observando qué ficheros y clases *stubs* se van necesitando por su ejecución.

El *browser* carga los ficheros `.html` y `.class`. En el *log* del servidor aparecen (la dirección IP es una cualquiera):

```
11.0.0.1   ClientApplet.html
11.0.0.1   ClientApplet.class
```

A partir de aquí, y como resultado de la inicialización del *applet*, se descargan las clases necesarias del ORB, además de las clases *stub*, `Helper` y `Holder` que ha generado el compilador de IDL:

```
11.0.0.1   Bank/AccountManagerHelper.class
11.0.0.1   Bank/AccountManager.class
11.0.0.1   Bank/_st_AccountManager.class
```

Además, hay un archivo que llama la atención y que servirá para ilustrar la siguiente sección, éste es `gatekeeper.ior`:

```
11.0.0.1   gatekeeper.ior
```

Esto responde al problema planteado en la sección 4.1.3: Una vez que el puente IIOP está funcionando, el *applet* puede actuar como un cliente CORBA normal. Sin embargo, antes de que el puente se establezca, ¿cómo obtiene un *applet* la referencia inicial al puente que le permite enrutar sus peticiones a través de él? La segunda parte que compone el `gatekeeper` es la pasarela IIOP. Ésta se implementa como un objeto CORBA del que el *applet* tiene que obtener una referencia. La referencia existe entonces como una cadena que representa a un IOR (*Interoperable Object Reference*) y que está guardado en el fichero llamado `gatekeeper.ior`. El *applet* puede recuperar ese fichero a través de HTTP, obtener la cadena que representa al IOR, e invocar al método del interfaz del ORB "`string_to_object()`" para transformar este IOR en una referencia real. En la siguiente sección analizaremos estas referencias más detenidamente, ya que nos servirán, como dijimos, para desprendernos de las características propietarias.

4.5 Programación avanzada

En esta sección trataremos una serie de tópicos algo más avanzados que el ejemplo simple de la sección anterior. Entre ellos está el uso de IORs para obtener referencias a objetos de forma estándar, el interfaz de invocación dinámica, el mecanismo de *Callback*, que permite a los clientes convertirse en servidores. También se verá como utilizar los servicios añadidos del ORB, como el servicio de nombrado (*Naming Service*) o el de *Trader*. Para finalizar, hacemos un repaso a qué servicios pueden ofrecer los dinstintos productos de manera que representen su competencia distintiva.

4.5.1 La cuestión de la *interoperatividad*

Una de las características más aclamadas de CORBA 2.0 es la posibilidad de integrar de forma estándar ORBs de distintos fabricantes. Como hemos visto a lo largo de este capítulo, esto se consigue a través de los protocolos GIOP y sus derivados, como IIOP. Como parte de estos protocolos, se ha estandarizado cómo una referencia a un objeto debe ser convertida a un *string* (a través del métodos del interfaz del ORB "`object_to_string()`") de manera que esta referencia identifique al objeto de forma universal. Estas referencias son lo que hemos venido llamando IORs. En el entorno Internet (en el que se utiliza el protocolo IIOP) esto significa que cada IOR debe contener la dirección del *host* donde reside su ORB y una referencia interna en ese ORB ([PWGB98, cap. 4], [VD98, cap. 5]). En esta sección veremos algún ejemplo de IOR, y desarrollaremos una aplicación que, dado un IOR, es capaz de conectarse de forma transparente a un objeto implementación.

4.5.1.1 IORs

En la sección anterior vimos cómo `gatekeeper` escribía el IOR del objeto que implementaba en un fichero. Este fichero es el siguiente:

```
IOR:000000000000002e49444c3a7669736967656e69632e636f6d2f676174656b6
5657065722f416c6961734d616e616765723a312e3000000000000020000000000
0000690001000000000000931312e302e302e3100003a980000005100504d4300000
0000000002e49444c3a7669736967656e69632e636f6d2f676174656b6565706572
2f416c6961734d616e616765723a312e3000000000000001149494f5020476174652
```

```
d4b6565706572000000000000001000000240000000000000010000000100000000
1400000000000000000000000000010109000000000
```

Los IORs tienen un tamaño variable, ya que en su interior llevan una lista de opciones llamadas "*Component profiles*". Como se ve, consta de los cuatro caracteres "`IOR:`" seguido de un conjunto de datos en hexadecimal. Con un pequeño programa en Perl podemos convertir a caracteres los valores hexadecimales:

```
# -*- Perl -*-

# Leer el IOR
$ior = <STDIN>;
# Separar la parte "IOR:"
$data = $' if $ior =~ /:/;
# Separar los pares de valores hexadecimales en caracteres
$data =~ s/[a-zA-Z0-9]{2}/pack("C",hex($&))/eg;
# Imprimir el resultado
print $data;
```

y con la línea de comandos

```
perl conv.pl <gatekeeper.ior | strings
```

(el comando strings devuelve las cadenas legibles de la salida) obtenemos:

```
.IDL:visigenic.com/gatekeeper/AliasManager:1.0
11.0.0.1
.IDL:visigenic.com/gatekeeper/AliasManager:1.0
IIOP Gate-Keeper
```

en el que la primera línea contiene el identificador del *Interface Repository* del objeto, mostrándonos que implementa el interfaz `AliasManager`, en su versión 1.0; la segunda línea contiene el nombre del *host* (que coincide con el presente en los *logs*); y la última línea contiene lo que podría ser una descripción. Una definición completa de los IORs se puede encontrar en el estándar CORBA ([OMG97, cap. 10]) y en ORBs que ofrecen el código fuente ([Chi98]).

4.5.1.2 Ejemplo de uso de IORs

En esta sección, finalmente utilizaremos un IOR para obtener una referencia que nos servirá para invocar procedimientos de forma remota. Para ello partiremos del IDL de la sección 4.4 y construiremos un servidor y un cliente. El servidor imprimirá el IOR del objeto creado, que será utilizado por el cliente para conectarse a la implementación remota. Es conveniente que los ficheros `.idl` se compilen utilizando algún *flag* de compatibilidad del compilador de IDL. En el caso de VisiBroker, la opción es `-portable` ó `-strict`. Así:

```
idl2java -strict Bank.idl
```

En cuanto al servidor, he aquí uno sencillo:

```
// -*- Java -*-

import org.omg.CORBA.*;

public class Server2
{
    public static void main(String[] args)
    {
    try {
        // Inicializar el ORB
        ORB orb = ORB.init(args,null);
        // Inicializar el BOA
        BOA boa = orb.BOA_init();

        // Crear un objeto Bank.AccountManager
        Bank.AccountManager manager =
            new AccountManagerImpl();
        // Exportar la referencia al BOA
        boa.obj_is_ready( manager );
        System.out.println( orb.object_to_string( manager ));

        // Esperar peticiones
        boa.impl_is_ready();
    }
    catch( SystemException e )
    {
        System.err.println(e);
    }
    }
}
```

éste realiza la misma labor que el visto en la sección anterior, solo que imprime el valor del IOR. Hay, sin embargo, un pequeño detalle: el BOA está definido en *pseudo-IDL*, lo que significa que sólo una parte de él está definido en IDL y la restante es dejada a la decisión del implementador. La especificación actual del BOA no es suficiente para Java. Por eso, el estándar del *mappint* de IDL a Java no lo contempla. Por tanto, la creación del BOA no es estándar. Sin embargo, los productos más importantes que implementan un ORB Java (VisiBroker, OrbixWeb) ofrecen su propio interfaz, con unas interfaces muy equivalentes, aunque no compatibles. Aún así, la adaptación a uno u otro producto es muy sencilla. Para conseguir una portabilidad total, podemos utilizar el método `connect()` que veremos en la sección 4.5.3.

Nótese, además, que el servidor ha invocado al constructor por defecto de la clase `AccountManagerImpl`. Éste no estaba definido, por lo que lo hemos tenido que escribir. Es tan sencillo como:

```
public AccountManagerImpl() { super(); }
```

Tras la compilación (con "`vbjc Server2.java`"), la ejecución con "`vbj Server2`" produce en la salida un IOR:

```
IOR:000000...
```

En cuanto al cliente, utilizaremos una aplicación, en vez de un *applet*. Suponemos que el IOR del objeto remoto se lo daremos en el primer argumento de la línea de comando. El cliente queda:

```
import org.omg.CORBA.*;

public class ClienteCORBA
{
    public static void main(String[] args)
    {
        try {
            // Inicializar el ORB
            ORB orb = ORB.init(args,null);

            // Obtener una referencia al objeto a partir del IOR
            org.omg.CORBA.Object obj = orb.string_to_object( args[0] );

            // Hacer un "casting" hacia el objeto que queremos
            Bank.AccountManager manager =
                Bank.AccountManagerHelper.narrow( obj );

            if (manager == null)
            {
                System.err.println("Referencia de tipo inadecuado.");
                System.exit( -1 );
            }

            // Obtener la cuenta del cliente pasado como parámetro
            Bank.Account acc = manager.open( args[1] );

            // Pedir el balance de la cuenta obtenida
            float balance = acc.balance();

            System.out.println("Cliente: " + args[1]);
            System.out.println("Balance: " + balance);
        }
        catch (SystemException e)
        {
            System.err.println(e);
        }
    }
}
```

Hay muchas cuestiones de interés en el cliente. Lo primero es que acepta dos parámetros en la línea de comandos. El primero es el IOR del objeto `AccountManager` al que conectarse. El segundo es el nombre de la cuenta de la que queremos saber el saldo. Estarán, respectivamente en `args[0]` y `args[1]`. Con este IOR, se invoca al método del ORB `string_to_object()`, que retorna un objeto genérico del tipo `org.omg.CORBA.Object`. Después, con la ayuda de la clase `AccountManagerHelper`, el objeto es convertido (*narrowed*) al tipo requerido. La conversión retornará un valor `null` si no es posible. A partir de aquí, la ejecución continúa como en el cliente *applet* de la sección anterior. **Nótese que, al igual que ocurría con el cliente *applet*, no hay inicialización del BOA: ésto sólo es útil en la parte de implementación**.

Es un tanto irrisorio suponer que el que ejecuta el programa cliente va a introducir a mano el IOR en la línea de comandos. Sin embargo, como no existe protocolo estándar para la transmisión inicial de las referencias, se deben idear otros medios, como por ejemplo, que el servidor mantenga un directorio accesible a través de HTTP y SSL, por ejemplo, con ficheros del tipo "`gatekeeper.ior`". Para solucionar esto, *Visigenic*, *IBM*, *Netscape*, *Oracle* y *SunSoft* han presentado una propuesta al OMG para un "Servicio de Nombres Interoperable" que se encargaría, entre otras cosas, de iniciar un protocolo de inicialización a través de IIOP para obtener las referencias iniciales a través de un puerto bien conocido TCP/IP. Sin embargo, este protocolo todavía tiene que ser admitido.

Lo que hemos hecho aquí es redirigir la salida del servidor hacia un fichero llamado `server.ior`:

```
vbj Server2 > server.ior
```

Entonces, de alguna manera mágica, podemos transportar ese fichero hasta la máquina cliente y entonces ejecutar el cliente con algo como esto:

```
vbj ClienteCORBA ‘cat server.ior‘ Diego
```

donde las comillas inversas sustituyen la salida del comando `cat` y se especifica la cuenta de "Diego". El cliente termina con una salida como:

```
Cliente: Diego
Balance: 425.66
```

El grado de independencia de Sistema Operativo y de localización que se consigue es asombroso. Por ejemplo, hemos estado utilizando la versión de VisiBroker para Win32, con sus utilidades `vbjc` para compilar y `vbj` para ejecutar (que por lo tanto, son programas específicos Windows). Sin embargo, ahondando un poco más en la teoría, podemos ver que lo único necesario a la hora de compilar y/o ejecutar programas CORBA en Java es el conjunto de clases Java que implementan el ORB y un compilador de Java normal. Así, podemos intentar ejecutar este ejemplo (que no depende de las características específicas de VisiBroker) en Linux, por ejemplo:

```
javac -classpath ../../lib/vbjorb.jar:.:$CLASSPATH Server2.java
```

en el que incluimos el fichero de librería Java `vbjorb.jar`, el ORB en Java de VisiBroker, además del resto de clases en `$CLASSPATH`. Igual se compilará el cliente. El resultado es que ambos se compilan sin problemas. La ejecución no es menos sencilla. Primero el servidor:

```
java \
   -classpath ../../lib/vbjorb.jar:.:../../lib/vbjapp.jar:$CLASSPATH \
   Server2 > server.ior &
```

que se inicia en segundo plano y se le hace que imprima el IOR en el fichero `server.ior`. Ahora el cliente:

```
java \
   -classpath ../../lib/vbjorb.jar:.:../../lib/vbjapp.jar:$CLASSPATH \
   ClienteCORBA ‘cat server.ior‘ Diego
```

obteniendo el mismo resultado. **¡Pero esta vez en Linux, y sin cambiar nada de código!**.

4.5.2 Invocaciones dinámicas

Como vimos en la introducción a CORBA, éste permite que las invocaciones a objetos se realicen de forma estática o dinámica, esto es, generadas en tiempo de ejecución. El interfaz del ORB ofrece los mecanismos necesarios para realizar invocaciones dinámicas de forma estándar. El proceso a seguir para realizar una invocación dinámica se puede resumir en los siguientes puntos:

1. **Obtener la referencia al objeto**, como siempre.
2. **Obtener información sobre el/los interfaces que implementa el objeto**. Esto se puede hacer con el método de la clase `org.omg.CORBA.Object` `_get_interface()`. Éste retorna un objeto del tipo `InterfaceDef`, del que, a su vez se puede obtener un `FullInterfaceDescription` con su método `describe_interface()`. Esto da acceso a toda la estructura del *Interface Repository*, no cubierta aquí (véase, sin embargo, [OMG97, cap. 7] y [OH97, cap. 19]).
3. **Construir un objeto `Request`** con los datos del método a invocar, parámetros de entrada y de salida, y valor devuelto. En términos de tipos de datos IDL, un `Request` consta de los siguientes:
   - `string`. El nombre de la operación.
   - `NamedValue`. El valor devuelto por la operación.
   - `NVList`. La lista de valores de parámetros.
4. **Invocar al método, a través del método `invoke()` de `Request`**.

La obtención de la referencia ya la vimos anteriormente. En cuanto a la búsqueda en el *Interface Repository*, en este caso no es necesaria, ya que sabemos qué métodos implementan nuestros objetos. Hay varias maneras de construir e invocar al objeto `Request`. Esto es porque el ORB ofrece varias maneras de crear listas `NVList` y el propio objeto `Request` posee métodos que permiten manejar la lista de parámetros, etc.

Aquí lo haremos de modo sencillo. Un ejemplo más complejo que comprueba los tipos de los parámetros y accede al *Interface Repository* se puede encontrar en [VD98, cap. 10]. Nótese que sólo tenemos que implementar el cliente, ya que el servidor no distingue entre invocaciones estáticas o dinámicas. He aquí el código del nuevo cliente:

```
import org.omg.CORBA.*;

public class ClienteCORBA_DII
{
    public static void main(String[] args)
    {
        try {
            // Inicializar el ORB
            ORB orb = ORB.init(args,null);

            // Obtener una referencia al objeto a partir del IOR
            org.omg.CORBA.Object obj = orb.string_to_object( args[0] );

            // Crear la petición al objeto 'obj'
            Request req = obj._request("open");

            // Establecer el tipo devuelto
            req.set_return_type(Bank.AccountHelper.type());

            // Establecer los argumentos
            org.omg.CORBA.Any nombre = req.add_in_arg();
            nombre.insert_string( args[1] );

            // Invocar la operación
            req.invoke();

            // Tener en cuenta si se ha lanzado alguna excepción
            java.lang.Exception ex = req.env().exception();
            if (ex != null)
            {
                throw (org.omg.CORBA.SystemException) ex;
            }

            // Obtener la cuenta del cliente pasado como parámetro
            Bank.Account acc;
            acc = Bank.AccountHelper.extract( req.return_value() );

            // Pedir el balance de la cuenta obtenida
            float balance = acc.balance();

            System.out.println("Cliente: " + args[1]);
            System.out.println("Balance: " + balance);
        }
        catch (SystemException e)
        {
            System.err.println(e);
        }
```

```
    }
}
```

Varias cosas: Un objeto `Request` se crea gracias al método `_request()` de `org.omg.CORBA.Object` aplicado al objeto del que hemos obtenido la referencia. El parámetro "`open`" indica el nombre de la operación. El tipo del parámetro devuelto (en este caso `Bank.Account`) es especificado gracias al método `set_return_type()` que acepta un `TypeCode`, en este caso devuelto por la clase `Helper` con su método `type()` (véase sección 4.3.3).

A continuación, introducimos el parámetro de entrada, a través del método `add_in_arg()`. Éste devuelve un `Any` al que se le puede asociar cualquier tipo. En este caso, le insertamos una cadena: el nombre del que queremos obtener la cuenta.

Finalmente, con la construcción `req.invoke()` invocamos la operación en el objeto. El resultado y las excepciones (posíblemente) generadas se guardan también en el objeto `Request`. El resultado, obtenido a través de `req.return_value()` es un `Any`. Se debe utilizar el método `extract()`, otra vez de una clase `Helper`; en este caso de `AccountHelper`. A partir de aquí, el código sólo invoca al método `balance()` para obtener el saldo de la cuenta recibida.

El lector puede obtener un buen tutorial de cómo realizar invocaciones dinámicas leyendo el código de los *stubs* y *skeletons* portables (sección 4.3.9). Los *stubs* portables realizaban sus llamadas a través del mecanismo estándar de la invocación dinámica. En el caso de VisiBroker estas clases tienen el prefijo "`_portable_stub_`".

4.5.3 El cliente se convierte en servidor: *Callbacks* distribuidos

Cuando un cliente desea ser informado por un servidor (en el sentido de que el servidor invoca a un método del cliente como parte de su funcionamiento), intercambiándose así momentaneamente los papeles de cliente y servidor, se dice que el cliente espera o recibe un *callback* o *llamada de respuesta*. El proceso es sencillo y se basa en que cualquier elemento conectado al ORB (ya sea cliente o servidor) puede crear objetos implementación de algún interfaz. Esta filosofía es la que se encuentra bajo la creación de aplicaciones distribuidas *peer-to-peer* en las que todos los elementos que populan el sistema pueden actuar en algún momento como clientes o como servidores.

Los pasos a seguir por un cliente que quiere recibir un *callback* son los siguientes:

1. **El cliente crea un objeto implementación de algún interfaz**. Este proceso es el mismo que hemos visto en los programas servidores.

2. **El cliente conecta este objeto con el ORB**. Esto es algo distinto a cómo los servidores que hemos visto lo hacen. Y esto es porque bajo los clientes no existe ningún *Object Adapter* de apoyo. El método `connect()` del interfaz del ORB se utiliza para conectar una implementación sobrepasando cualquier OA existente. Por ello, las interacciones inversas (es decir, de servidor a cliente) están restringidas a ser más sencillas.

3. **El cliente pasa una referencia del objeto creado al servidor**. De alguna manera, ya sea como parámetro de una invocación o a través de un IOR, el cliente debe informar al servidor de la referencia al objeto sobre el qué debe realizar el *callback*.

4. **El servidor invoca un método del objeto implementado por el cliente (se realiza el *callback*)**. Como resultado de su funcionamiento, el servidor llama al cliente para indicar algún suceso cuando convenga o para completar su funcionalidad propia.

Para ilustrarlo, supongamos un cliente CORBA que participa en un sistema de procesamiento en tiempo real y que muestra una lista de precios. El cliente quiere ser informado en tiempo real cada vez que el precio para un producto cambie. Podemos modelar el escenario con el siguiente conjunto de interfaces IDL:

```
// -*- IDL -*-

module OnLine
{
    interface Listener
    {
        void updatePrice( in string productId, in float newPrice );
    };

    // Manager de clientes
    interface Manager
    {
        void register( in OnLine::Listener newListener );
        void deregister( in OnLine::Listener listener );

        void updateDependants( in string productId, in float newPrice );
    };

    // Resto de los interfaces
    // ...
};
```

El servidor implementaría el interfaz `OnLine::Manager`, y el cliente el `OnLine::Listener`. Cuando un cliente es lanzado, se debe registrar en el `Manager`. Éste, a su vez, cuando el precio de algún producto cambia, le informa a través del método `updatePrice()` de `Listener`. Cuando el cliente deja de estar activo, se debe borrar del registro. El método `updateDependants()` puede ser invocada tanto por los clientes (con lo que todos ellos pueden actualizar el precio del producto) o puede ser generado por eventos en la parte servidor o en otros clientes. El resultado es la actualización de todos los clientes registrados[‖].

La implementación de `Manager` debe mantener internamente una lista con los `Listener` registrados. Esto se puede hacer como un vector:

```
// -*- Java -*-
package OnLine;
```

[‖]Si no se requiere (o no se desea) que los clientes puedan actualizar la lista de precios, el diseño debería cambiar.

```
import java.util.Vector;
import org.omg.CORBA.*;

public class ManagerImpl extends OnLine._ManagerImplBase
{
    private Vector _listeners = new Vector();

    public ManagerImpl() { super(); }

    public void register( OnLine.Listener newListener )
    {
        _listeners.addElement( newListener );
    }

    public void deregister( OnLine.Listener listener )
    {
        _listeners.removeElement( listener );
    }

    public void updateDependant ( java.lang.String id, float price )
    {
        for (int i = 0; i < _listeners.size() ; i++)
        {
            OnLine.Listener l = (OnLine.Listener)_listeners.elementAt(i);
            l.updatePrice( id, price );
        }
    }
}
```

El servidor lanzará de manera normal una implementación de este interfaz. Pero lo diferente aquí es cómo el cliente logra generar un objeto implementación para quedar actualizado por el servidor sin disponer de un BOA para registrarlo. La respuesta a esto es el método `connect()` del interfaz del ORB. La siguiente clase tiene dos roles: actúa como cliente en sí (con la función estática `main()`); e implementa el interfaz `OnLine.Listener` como parte de su implementación (heredando de `OnLine._ListenerImplBase`):

```
// -*- Java -*-
import java.util.Hashtable;
import org.omg.CORBA.*;

public class ClienteListener extends OnLine._ListenerImplBase
{
    // Método estático 'main'
    public static void main(String[] args)
    {
        try {
        // Inicializar el ORB
```

```
        ORB orb = ORB.init( args , null );

        // Obtener (mágicamente) una referencia a un
        // OnLine.Manager (véanse secciones anteriores)
        OnLine.Manager manager = // ...

        // Crear un objeto Listener
        ClienteListener client = new ClienteListener();

        // Conectarlo con el ORB
        orb.connect( (OnLine.Listener)client );

        // Registrar el cliente
        manager.register( (OnLine.Listener)client );

        // Iniciar el cliente
        client.run();
        } catch (SystemException ex) {
            System.err.println(e);
        }
    }

    // Funcionalidad del cliente
    public void run() { ... }

    // Implementación de la funcionalidad de "OnLine.Listener"
    public ClienteListener() { super(); }

    public void updatePrice( java.lang.String id, float price )
    {
        // Actualizar la variable '_productos' y el GUI
        // ...
    }

    // Datos privados de la implementación
    private Hashtable _productos;
}
```

No se muestra la parte del GUI. El cliente tiene tres partes: el método `main()` que inicia el ORB y crea la instancia "`client`". Esta instancia actúa como objeto implementación de `OnLine.Listener` y como objeto de la clase `ClinenteListener` que implementa la funcionalidad del cliente. A través del *narrowing* se consigue invocar al método `register()` del `maganer` con esa misma instancia vista como implementadora del interfaz `OnLine.Listener`. La alternativa a esta construcción habría sido la delegación ([VD98, cap. 13]).

4.5.4 CORBAServices y CORBAFacilities

Esta sección trata de ser un pequeño ejemplo de cómo pueden ser utilizados los servicios añadidos al ORB en forma de IDL. Estos servicios están definidos como parte de los CORBAServices y CORBAFacilities. Aunque de cara al ORB todos ellos presentan un conjunto de interfaces, existen una serie de programas de apoyo específicos de la implementación concreta del ORB (como *demonios* y etcéteras) de los que también veremos su utilización. Esto no significa que al utilizar los servicios de un vendedor específico quedemos ligados a él: el OMG ha definido muy bien estos servicios de forma estándar, por lo que el código puede ser transportado sin casi modificación entre distintos ORBs. Además, siempre queda la posibilidad de la interoperatividad entre ORBs.

El conjunto de servicios estándar es grande y sigue creciendo. Entre ellos podemos nombrar:

Nombrado El servicio de nombres permite ligar objetos a nombres de forma relativa a un contexto jerárquico.

Eventos Con el servicio de eventos se permite que distintos objetos se registren como interesados en distintos eventos que se producen en el sistema, bien lanzados por algún objeto como parte de su funcionalidad o de forma espontánea. El diseño de este servicio no requiere de ningún servidor centralizado, y está especialmente adaptado para ambientes distribuidos.

Ciclo de Vida Provee servicios de copia, movimiento y eliminación de grafos de objetos relacionados.

Servicio de Persistencia Provee servicios de mantenimiento de objetos persistentes. El diseño permite que varias implementaciones coexistan en un mismo ambiente, ya que es posible, por ejemplo, que los requerimientos de almacenamiento para guardar documentos no sean los mismos que para guardar Bases de Datos u objetos en un OODBMS.

Relaciones Permite establecer relaciones entre objetos. En particular, se añaden los tipos de objetos "relación" y "rol". Un *rol* representa a un objeto que toma parte en una relación. Se pueden añadir atributos específicos de roles y de relaciones, además de cardinalidades, etc. que serán comprobadas en tiempo de ejecución.

Externalización Este es el equivalente a la "Serialización" Java. Ofrece estándares e interfaces que permiten convertir a objetos en *streams* de datos y viceversa.

Transacciones El servicio de transacciones ofrece a los objetos la semántica de una transacción atómica: las acciones entre el inicio y el final de la transacción se realizarán todas o no se realizará ninguna.

Control de la Concurrencia Este servicio permite a varios clientes organizar su acceso concurrente a recursos compartidos.

Licencias El servicio de Licencias permite a los productores controlar el uso de su propiedad intelectual.

Consulta El servicio de consulta permite realizar consultas en conjuntos de objetos. Estas consultas están realizadas en un lenguaje declarativo y pueden indicar valores de atributos, invocar operaciones, etc.

Propiedades Permite asociar a los objetos un conjunto de propiedades que pueden ir modificándose en tiempo de ejecución. Las propiedades son conjuntos de pares $< nombre, valor >$. Los nombres son cadenas de caracteres, y los valores, objetos del tipo `Any` de CORBA.

Seguridad Todos los temas relacionados con la seguridad, incluyendo identificación, autentificación, autorización, encriptación, etc.

Tiempo Permite a un conjunto de objetos obtener el tiempo junto con un error asociado. Esto permite ordenar eventos que se producen en el sistema.

Colecciones Provee un mecanismo uniforme de creación y manejo de colecciones de objetos.

Trading Ofrece un servicio de páginas amarillas para los objetos, en el que se puede buscar objetos por funcionalidad, por tipos de servicios, calidad de servicio (QOS), etc. Este servicio es uno de los más flexibles y potentes actualmente disponibles por algunas implementaciones.

Esto da una idea del tamaño y ámbito de los servicios definidos para los objetos y, en general, del esfuerzo que el OMG está realizando para conseguir un conjunto de estándares de calidad.

Es obvio que la documentación de todos estos servicios sobrepasa el alcance de este proyecto. No obstante, en esta sección veremos, de forma genérica, cómo se obtienen las referencias iniciales a estos servicios y cómo se utiliza (de forma rudimentaria) el servicio de nombrado.

4.5.4.1 Obteniendo las referencias iniciales a servicios

CORBA 2.0 provee un mecanismo estándar de obtención de referencias a objetos que implementan los servicios descritos anteriormente. Como hemos advertido algunas veces, a través de los métodos del interfaz del ORB `list_initial_services()` y `resolve_initial_references()`. Un ejemplo de su uso podría ser el siguiente:

```
java.lang.String[] servicios = orb.list_initial_services();
for (int i = 0;i< servicios.length; i++)
    System.out.println( servicios[i] );
```

que imprime una lista como la siguiente:

```
ChainClientInterceptorFactory
NameService
ChainServerInterceptorFactory
URLNamingResolver
HandlerRegistry
ChainBindInterceptor
```

De ellos, en la siguiente sección siguiente nos interesará el "`NameService`". Los restantes son propietarios o sirven para otros fines. La bibliografía dará cuenta de ellos ([PWGB98], [VIS97b], [VD98]).

4.5.4.2 Un ejemplo: CORBA *Naming Service* (*CosNaming*)

El servicio de nombres responde a la pregunta: ¿cómo puede un objeto obtener referencias a otros objetos de forma estándar y sin tener que utilizar los inconvenientes IORs? Éste permite localizar referencias de objetos por nombre. El servicio permite organizar *contextos de nombrado* (*Naming Contexts*) de forma jerárquica, conectando, si se desea, varios ORBs departamentales, de empresa, etc.

La teoría que rodea a cada servicio es realmente voluminosa. Aquí se presentará un ejemplo simple. Ejemplos más elaborados se pueden encontrar en [PWGB98, cap. 15] y en [VD98, cap. 8].

El servicio de nombrado ofrece algunos interfaces. El más importante es el `NamingContext`. Éste permite ligar (*bind*) objetos con nombres (de forma parecida a como lo hacíamos en la sección 4.4 con el mecanismo propietario de VisiBroker, pero esta vez de forma estándar) y recuperar objetos por nombre (*resolve*).

El ejemplo que hemos preparado es muy simple: consta de un cliente que crea un objeto implementación y lo liga al ORB (como se vio en la sección 4.5.3), pero que utiliza el servicio de nombrado para, primero registrar, y después comprobar que el objeto ha sido registrado. Normalmente estas dos partes estarían en el servidor y el cliente respectivamente. El cliente obtiene inicialmente una referencia a un `NamingContext` a través de la resolución de referencias iniciales. Este contexto se denomina *contexto raíz* (*root context*). En este contexto, registrará el objeto creado (del tipo `Bank.AccountManager`) dándole un nombre:

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

public class ClienteNaming
{
    public static void main(String[] args)
    {
        try {
            // Inicializar el ORB
            ORB orb = ORB.init(args,null);

            // Obtener las referencias a servicios iniciales
            String[] servicios = orb.list_initial_services();
            for (int i = 0;i<servicios.length ;i++)
                System.out.println(servicios[i]);

            // Obtener el NamingContext raíz
            org.omg.CORBA.Object obj =
                orb.resolve_initial_references("NameService");
            NamingContext rootContext =
                NamingContextHelper.narrow( obj );

            if (rootContext == null)
            {
```

```
                System.err.println("No hay Naming Service!");
                System.exit( -1 );
            }

            // Crear un nuevo objeto y ligarlo con un nombre
            Bank.AccountManager manager =
                (Bank.AccountManager)new AccountManagerImpl();

            // Conectar con el ORB
            orb.connect( manager );
            System.out.println( orb.object_to_string(manager));

            // Ligar con un nombre
            NameComponent[] AccountManagerName = new NameComponent[1];
            AccountManagerName[0] = new NameComponent("Account Manager","");
            rootContext.rebind(AccountManagerName,manager);

            // Intentar obtenerlo desde el rootContext
            org.omg.CORBA.Object obj2 =
                rootContext.resolve(AccountManagerName);

            // Comprobar si ambas referencias son iguales
            System.out.println (orb.object_to_string(obj2));
        }
        catch (Exception e)
        {
            System.err.println(e);
        }
    }
}
```

El programa obtiene una referencia inicial al servicio de nombres a través del método `resolve_initial_references()`. Esta referencia, gracias al método estático `narrow()` de `NamingContextHelper`, se puede transformar a una referencia a un `NamingContext`. Tras crear el objeto implementación y conectarlo con el ORB, se imprime su IOR. Éste servirá para comprobarlo después. El siguiente paso es construir un nombre para el objeto. El servicio de nombres requiere un array de `NameComponent`s, elementos que contienen un "nombre" y un "tipo" (no como los tipos de ningún lenguaje ni de IDL, sino de forma genérica). Los distintos componentes del array están separados por "`/`", y especifican cada vez un nivel más interno de la jerarquía, igual que la estructura de directorios de UNIX. Para nuestro objeto sólo utilizaremos un elemento `NameComponent` y de él, sólo utilizaremos el nombre: "`Account Manager`". Con el método `rebind()` de `NamingContext` asociamos el objeto `manager` con su lista de `NameComponent`s.

La última parte del programa utiliza el array de `NameComponent` utilizado que nombró al objeto para localizarlo ahora con el método `resolve()`. Éste retorna una referencia, cuyo IOR también se imprime. Como esperábamos **¡las dos referencias son iguales: es el mismo objeto!**:

```
ChainClientInterceptorFactory
NameService
ChainServerInterceptorFactory
URLNamingResolver
HandlerRegistry
ChainBindInterceptor

IOR:000000000000001c49444c3a42616e6b2f4163636f756e744d616e61
6765723a312e3000000000020000000000000048000100000000000093131
2e302e302e3100000429000000300050 4d43000000010000001c49444c3a
42616e6b2f4163636f756e744d616e616765723a312e300000000001ad33
81e8000000010000002400000000000000010000000100000014000000000
0000000000000000001010900000000

IOR:000000000000001c49444c3a42616e6b2f4163636f756e744d616e61
6765723a312e3000000000020000000000000048000100000000000093131
2e302e302e3100000429000000300050 4d43000000010000001c49444c3a
42616e6b2f4163636f756e744d616e616765723a312e300000000001ad33
81e8000000010000002400000000000000010000000100000014000000000
0000000000000000001010900000000
```

La ejecución de este programa no es tan sencilla, por lo menos en el entorno VisiBroker. En primer lugar debemos instalar el *Visibroker Naming Service*, un producto vendido aparte. Antes de ejecutar el cliente, hay que ejecutar el *Smart Agent*, ya que las clases que implementan el servicio de nombres lo utilizan: es (por desgracia) un requisito ineludible. Después hay que iniciar el servicio de nombres dándole un contexto raíz. Esto se puede hacer con la siguiente línea de comando:

```
start vbj -DORBservices=CosNaming -DJDKrenameBug \
  com.visigenic.vbroker.services.CosNaming.ExtFactory ROOT root.log
```

lo que (en Windows 95/NT) inicia el servicio de nombres con el *root context* llamado "ROOT". El IOR del servicio se escribirá en el fichero "`root.log`"**.

El programa cliente se puede ejecutar entonces indicándole a la máquina virtual Java que dispone del servicio de nombres:

```
vbj -DORBservices=CosNaming -DSVCnameroot=ROOT ClienteNaming
```

El primer parámetro indica que el servicio de nombres está activo. El segundo, indica el nombre del *root context*. Este nombre será utilizado por las clases del ORB para localizar al objeto raíz `NamingContext` a través del *Smart Agent*.

4.5.5 Qué ofrecen los distintos productos

La labor del OMG es una de las ma's apreciables de toda la historia de la informática. Como hemos visto, sus estándares nos permiten lograr una independencia del fabricante y una interoperatividad entre plataformas, lenguajes que antes no era imaginable.

**¡Y la opción `-DJDKrenameBug` corrige un error en el JDK de *Sun*!

Sin embargo, los distintos fabricantes incluyen variaciones sobre los estándares que hacen más sencilla de cara al programador alguna u otra tarea. Esto, aunque acelera y facilita el proceso de desarrollo con ese producto en particular, nos elimina la portabilidad entre productos. Las variaciones, a menudo son presentadas incluso en los primeros programas de ejemplo y en todos los tutoriales. En la documentación de cada producto (y en la de los estándares CORBA), hay que ahondar mucho para distinguir qué característica es proporcionada por el estándar o por el fabricante. Esto es un punto en contra de las distintas implementaciones.

Por ejemplo, el servicio de nombrado simple que ofrece VisiBroker a través de su *Smart Agent* y una serie de métodos no estándar (`bind()`, etc.) no es compatible con el de OrbixWeb de *IONA*, así como tampoco el *mapping* del *Basic Object Adapter*.

Pero ¿cómo pueden conseguir los distintos productos una competencia distintiva? La respuesta va en dos direcciones. Por desgracia, una es la inclusión de funciones no estándar que hagan de alguna manera más sencillo el proceso. Otra es la incorporación de más servicios y mejor integrados con el entorno. Por ejemplo, OrbixWeb ofrece servicios de eventos, *trading*, transacciones, etc., mientras que VisiBroker no ofrece el de transacciones, etc. Entre estos servicios también se incluyen herramientas más sofisticadas de depuración, de CASE integrado, *frameworks*, etc.

4.6 Fnorb: Un ORB implementado en Python

Hasta ahora, aunque hemos establecido en la sección 4.5.1 las bases para que varios ORBs de distintos fabricantes que implementen CORBA 2.0 trabajen juntos, no lo hemos llevado a la práctica. El primer ejemplo que presentamos aquí conecta un objeto implementación escrito en Java con un cliente escrito en Python, el primero ejecutándose en el entorno Win32, y el segundo ejecutándose en Linux. En el segundo ejemplo, el servidor estará escrito en Python y utilizaremos el cliente Java que poseemos. Así, se logra independencia de localización (a través del uso de IORs), de lenguaje de implementación y de hardware+Sistema Operativo.

Fnorb es un ORB que implementa CORBA 2.0 escrito completamente en Python ([Chi98]). Esto nos permite escribir tanto clientes como servidores que, a través de IDL, tendrán un interfaz CORBA estándar y podrán ser utilizados por objetos implementados en otros lenguajes y ejecutándose en otras plataformas. Su código consta de unos 370 Kilobytes de código Python, e implementa un servicio de nombrado compatible con *CosNaming* y un *mapping* propietario[††] aunque sencillo de IDL a Python.

Nuestro objetivo es ejecutar el servidor de la sección 4.5.1 que nos proporciona un IOR. Ese IOR se pasará en forma de fichero a un cliente escrito en Python. El cliente tendrá la misma funcionalidad que el correspondiente de la misma sección, pero esta vez estaremos comunicando implementaciones en Java y Python.

El proceso seguido es simple: Necesitamos el IDL que introdujimos en la sección 4.4 para producir los *stubs* y *skeletons* necesarios para Python. Esto se consigue con la siguiente línea de comandos:

```
fnidl Bank.idl
```

donde fnidl es el compilador de IDL de Fnorb. Éste genera dos directorios correspondientes al módulo `Bank`: `./Bank` y `./Bank_skel`. El primero contiene a las clases que implementan

[††]Ya que el OMG no lo ha definido

los *stubs*. El segundo, a las que implementan los *skeletons*. El programa cliente tendrá que importar los *stubs*. Su listado se muestra a continuación:

```
#!/usr/bin/python -O  # -*- Python -*-

""" Cliente para el banco. """

# Módulos 'standard' o 'built-in'.
import sys

# Módulos Fnorb.
from Fnorb.orb import CORBA

# Stubs generados por 'fnidl'.
import Bank


def main(argv):
    """ Cliente! """

    print 'Inicializando el ORB...'

    # Iniciar el ORB.
    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)

    # Leer el IOR de un fichero
    f = open('bank.ior', 'r')
    ior = f.read()
    f.close()

    # Convertir el IOR a una referencia de un objeto activo.
    manager = orb.string_to_object(ior)

    # Comprobar que el objeto es del tipo que esperamos
    if not manager._is_a('IDL:Bank/AccountManager:1.0'):
        raise 'Este no es un servidor "Banco"!'

    # Obtener la cuenta del servidor y el balance
    account = manager.open( argv[1] )
    balance = account.balance()
    print "Cliente: ", argv[1]
    print "Balance: ", balance

    return 0

################################################################
```

```
if __name__ == '__main__':
    # Ejecutar el cliente
    sys.exit(main(sys.argv))

######################################################################
```

El programa, aparte de incluir los módulos necesarios de Fnorb, importa el módulo "Bank". La función `main()` implementa toda la funcionalidad del cliente. En primer lugar, inicializa el ORB. Después lee del fichero "`bank.ior`" el IOR que contiene una referencia a un objeto `AccountManager`. La clásica utilización de `string_to_object()` transforma a la cadena en un objeto activo. Nótese que no hay "casting": en Python el tipado es dinámico, al estilo de Smalltalk.

Tras comprobar que la referencia indica a un objeto objeto pertenece a la clase `Bank/AccountManager`, se procede a invocar los métodos de manera habitual. Primero, el método `open()` sobre `manager` retorna un objeto, `account`, de tipo `Account`. Al objeto retornado, le aplicamos el método `balance()` para obtener su saldo. Finalmente, si el ámbito de nombres (o *namespace*), llamado en Python "`__name__`", es "`__main__`" (esto es, el programa se ha invocado directamente), se ejecuta la función `main()`. La salida obtenida es algo como esto:

```
Inicializando el ORB...
Cliente: Diego
Balance: 206.070007324
```

El segundo ejemplo requiere de una implementación en Python de la funcionalidad de los interfaces `AccountManager` y `Account`, además de un servidor que lance un objeto implementación de cada una de estas clases. A pesar de estar escrito en otro lenguaje distinto a Java, la estructura de todo el código que veremos a continuación es similar al que hemos ido analizando, al estar por debajo la teoría relativa a CORBA. En primer lugar, la implementación de la clase `Account` es muy sencilla. Se hace a través de la clase `AccountImpl`:

```
# -*- Python -*-

import Bank
import Bank_skel

class AccountImpl(Bank_skel.Account_skel):
    def __init__(self, initialBalance):
        self._balance = initialBalance

    def balance(self):
        return self._balance
```

que hereda de la clase `Account_skel` del paquete `Bank_skel`. El método `__init__()` es el constructor de la clase.

La implementación de `AccountManager` (a través de la clase `AccountManagerImpl`) es algo más compleja. Y esto es así porque debe añadir más objetos que implementan el interfaz `Account` a medida que se realizan peticiones `open()`. El listado se muestra a continuación:

```
# -*- Python -*-

from Fnorb.orb import CORBA,BOA

import Bank
import Bank_skel

from time import time
from random import randint

from AccountImpl import *

class AccountManagerImpl(Bank_skel.AccountManager_skel):

    # Constructor
    def __init__(self):
        # Llamar al constructor de la clase base
        CORBA.Object_skel.__init__(self)
        self._accounts = {}

    # Implementar operaciones
    def open(self, name):
        """ Implementa el método Bank/AccountManager/open """

        # Ver si el cliente ya se ha creado
        if self._accounts.has_key(name):
            return self._accounts[name]

        # No se ha creado, crear
        acc = AccountImpl( randint(0,10000) / 100.0 )
        boa = BOA.BOA_init()
        ref = boa.create('dsr2',AccountImpl._FNORB_ID)
        boa.obj_is_ready( ref, acc )

        # Y añadirlo al diccionario local (o a la base de datos)
        self._accounts[name] = acc
        return acc
```

Siguiendo el patrón de la anterior, ésta hereda de `Bank_skel.AccountManager_skel`. Utiliza el diccionario `_accounts` para guardar la asociación entre nombres de clientes y cuentas. El proceso de creación de un objeto implementación tiene en `Fnorb` dos etapas. Primero se crea una referencia a un objeto del interfaz requerido con el método `create()` del BOA. A este método se le envía una clave del objeto y una identificación del *Interface Repository*, que se encuentra en el atributo `_FNORB_ID` de todas las clases generadas a partir de IDL. A continuación se crea un objeto implementación, instanciando la clase `AccountImpl`. Por último, se liga a la referencia con el objeto utilizando el método del BOA `obj_is_ready()`

de dos argumentos.

Por último, el servidor debe crear un objeto de la clase que implementa el interfaz `AccountManager` y ligarlo al ORB, dando su referencia como un IOR guardado en un fichero llamado "`bank.ior`". La última línea, a través del método `_fnorb_mainloop()` entra en un bucle activo que espera a recibir todas las peticiones de los clientes. El listado se muestra a continuación:

```
#!/usr/bin/python -O

""" Servidor de 'Banco'. """

import sys
from Fnorb.orb import CORBA,BOA

import Bank
import Bank_skel
from AccountManagerImpl import *
from AccountImpl import *

def main(argv):
    print 'Inicializando el ORB...'
    # Inicializar el ORB.
    orb = CORBA.ORB_init(argv, CORBA.ORB_ID)

    print 'Inicializando el BOA...'
    # Inicializar el BOA.
    boa = BOA.BOA_init(argv, BOA.BOA_ID)

    print 'Creando la referencia a un objeto'
    # Crear una referencia ('dsr' es la clave del objeto).
    obj = boa.create('dsr', AccountManagerImpl._FNORB_ID)

    print 'Creando implementación...'
    # Crear una instancia del objeto implementación.
    impl = AccountManagerImpl()

    print 'Activando la implementación...'
    boa.obj_is_ready(obj,impl)

    f = open('bank.ior', 'w')
    f.write(orb.object_to_string(obj))
    f.close()

    print 'Servidor creado y aceptando peticiones...'

    # Comenzar el ciclo de recepción de eventos.
    boa._fnorb_mainloop()
```

```
    return 0


############################################################

if __name__ == '__main__':
    sys.exit( main(sys.argv) )

############################################################
```

Al estar implementado en Python, un lenguaje basado en compilación a *bytecodes* como Java, las ventajas de portabilidad que se pueden obtener con él se asemejan a las de este último (véase sección 4.1.1.1).

Capítulo 5

Conclusiones

En los dos capítulos anteriores, hemos estudiado las tecnologías más utilizadas actualmente para el desarrollo de aplicaciones distribuidas. En este capítulo presentamos un resumen comparativo de las características que cada tecnología presenta, que nos servirá para evaluarlas en base a los criterios que introducimos en el capítulo 2. De forma resumida, las características que se considerarán se engloban en cinco aspectos: 1) El proceso de desarrollo Cliente/Servidor; 2) el grado de integración con Java; 3) instalación y *deployment*; 4) rendimiento; y 5) Escalabilidad. Las distintas tecnologías estudiadas son *Sockets*, HTTP/CGI (y *Servlets*), RMI, DCOM y la integración Java/CORBA.

Esta comparativa está basada en [OH97, cap. 15] y aparece en la tabla de la página siguiente. Los puntos que se consideran en la tabla son los siguientes:

Nivel de Abstracción. A medida que el nivel de abstracción ofrecido por la tecnología aumenta, nuestra aplicación debe realizar menos tareas. Con los *Sockets* teníamos que definir convenciones de paso de parámetros, tipos de datos, *marshaling*, definición de servicios, etc. CORBA, DCOM y RMI poseen un mayor nivel de abstracción. HTTP/CGI está en el punto medio.

Integración con Java. CORBA y RMI proveen la mejor integración con Java, ya que DCOM lo ha integrado *a posteriori*. HTTP/CGI y *Sockets* ofrecen librerías de bajo nivel.

Soporte multiplataforma. CORBA, HTTP/CGI y *Sockets* se ejecutan en la mayoría de las plataformas. DCOM sólo soporta Windows 95/NT, y sólo con la máquina virtual Java de *Microsoft*.

Implementación total en Java. Hay implementaciones de CORBA, *Sockets*, RMI y HTTP/CGI escritas completamente en Java. Sin embargo, DCOM está escrito en C/C++ y se proveen adaptadores para DCOM sólo en su máquina virtual, en ninguna otra.

Soporte de tipos Las tres tenologías basadas en Objetos Distribuidos proveen soporte para tipos de datos y chequeo en tiempo de compilación y ejecución. CORBA y DCOM permiten especificar parámetros de salida y de entrada/salida. RMI sólo de entrada. HTTP/CGI provee un mínimo soporte de parámetros a través de MIME, y los *Sockets* no proveen ninguno.

Característica	CORBA	DCOM	RMI	HTTP/CGI	Sockets [a]
Nivel de abstracción	★★★★	★★★★	★★★★	★★	★
Integración con Java	★★★★	★★★	★★★★	★★	★★
Soporte multiplataforma	★★★★	★★	★★★★	★★★★	★★★★
Implementación total en Java	★★★★	★	★★★★	★★★★	★★★★
Soporte de tipos	★★★★	★★★★	★★★★	★	★
Faciliad de configuración	★★★	☆	★★★	★★★	★★★
Invoc. de métodos distribuida	★★★★	★★★	★★★	☆	☆
Estado entre invocaciones	★★★★	★★★	★★★	☆	★★
Meta-información	★★★★	★★★	☆	☆	☆
Invocaciones dinámicas	★★★★	★★★★	★	☆	☆
Rendimiento (*ping*)	★★★★	★★★	★★★	☆	★★★★
	3.3 ms	3.9 ms	5.5 ms	603.8 ms	2.0 ms
Seguridad	★★★★	★★★★	★★★	★★★	★★★
Transacciones	★★★★	★★★	☆	☆	☆
Objetos persistentes	★★★★	★	☆	☆	☆
Soporte multilenguaje	★★★★	★★★	☆	★★★	★★★★
Protocolo común multilenguaje	★★★★	★★★★	☆	★★★★	☆
Escalabilidad	★★★★	★	★	★★	★★★★
Estándar abierto	★★★★	★★	★★	★★★★	★★★★

[a] ★★★★ = mejor; ★ = peor; ☆ = N/A, muy mala o inexistente.

Tabla 5.1: Comparación de todas las tecnologías estudiadas.

Facilidad de configuración. La configuración de DCOM es desmoralizadora. Por comparación, todo lo demás parece fácil. No obstante, el trabajo con sistemas distribuidos no es fácil.

Invocación de métodos distribuida. Sólo las tres tenologías basadas en Objetos Distribuidos permiten la invocación distribuida de métodos. CORBA además soporta referencias a objetos únicas (al contrario de DCOM).

Estado entre invocaciones. El principal problema con HTTP/CGI es que no conserva el estado entre invocaciones. Con *Sockets* podemos mantener el estado, pero una vez más, esto va por cuenta del programador. Las tres tecnologías de Objetos Distribuidos conservan el estado de sus objetos, pero sólo CORBA ofrece un identificador único a cada objeto, que permite reconectar al mismo objeto (con su mismo estado) en un momento posterior.

Soporte de Meta-información. Sólo CORBA y DCOM soportan introspección. Ambos permiten descubrir de forma dinámica los interfaces que ofrece un objeto. CORBA soporta Repositorios de Interfaces inter-ORB, y DCOM sólo de forma local a través de librerías de tipos.

Invocación dinámica. Una vez más, sólo CORBA y DCOM ofrecen esta posibilidad.

Rendimiento. CORBA y los *Sockets* son los que ofrecen un mejor rendimiento, mientras que HTTP/CGI está *muy* lejos de estas cifras.

Seguridad. Los *Sockets* proveen seguridad a través de SSL (*Secure Socket Layer*). HTTP, también a través de S-HTTP y SSL. DCOM se basa en el sistema de seguridad de Windows NT. CORBA define también un servicio de seguridad, integrado en el ORB. También soporta el uso de SSL. RMItambién soporta SSL.

Transacciones. Sólo DCOM y CORBA soportan transacciones, este último a través del OTS (*Object Transaction Service*).

Objetos persistentes. Sólo CORBA permite objetos persistentes que también poseen referencias persistentes. DCOM ofrece un parche llamado *moniker*. Los *monikers* pueden asociar a un objeto con un contexto, aunque no dejan de ser un parche.

Soporte multilenguaje. Todos los lenguajes soportan *Sockets*, y la mayoría HTTP/CGI. RMI sólo funciona con Java. CORBA y DCOM ofrecen soporte multilenguaje. El primero define un estándar de *mapping* de alto nivel. DCOM, por contra, ofrece un estándar binario soportado por los compiladores de *Microsoft*.

Protocolo común multilenguaje. DCOM y CORBA proveen protocolos estándar de comunicación entre entidades independiente del lenguaje, el Sistema Operativo y la plataforma hardware. HTTP/CGI ofrece este soporte basado en los tipos MIME, que no deja de ser limitado.

Escalabilidad. *Sockets* soportan escalabilidad a través de redes TCP/IP. Sin embargo, este soporte es a un nivel de abstracción muy bajo. En contraste, CORBA ofrece federaciones de ORBs, dominios de nombres, etc. IIOP establece una manera estándar de propagar transacciones, seguridad, etc., a través de ORBs de distintos fabricantes. La idea básica es que provee la infraestructura para conectar ORBs débilmente acoplados.

Estándar abierto. Una infraestructura de aplicaciones distribuidas a escala mundial que posíblemente se integre con Internet no puede pertenecer a una única compañía. Debe basarse en un conjunto de estándares bien definido en el que los distintos vendedores puedan competir, sin alcanzar ninguno el control total. CORBA, *Sockets* y HTTP/CGI son abiertos: están controlados por un cuerpo de estandarización. DCOM y RMI no lo son.

Como se desprende de la tabla, en la que se analizan los puntos más importantes que hemos ido descubriendo durante el estudio de todas las tecnologías, la mejor tecnología es la integración de Java con CORBA.

Apéndice A

Listado de los programas

En este apéndice se listan los programas que se han desarrollado como ejemplo en este proyecto. No se incluyen los del capítulo dedicado a Java y CORBA ni los de DCOM porque ya fueron listados en sus correspondientes secciones.

A.1 Aplicación de charla con Sockets

A.1.1 ChatApplet.html

Esta es la página HTML que incluye al *applet* que actúa como cliente en la aplicación de *charla*:

```
<html>
<head>
<title>ChatApplet</title>
</head>
<body>
<center>
<h1>ąMi propio Chat!</h1>
</center>
<font size=-3> <p align=right>(c) DSR 1998</p></font>
<center>
<hr>

<applet
    code=ChatApplet.class
    id=ChatApplet
    width=600
    height=300 >
    <param name=direccion value="127.0.0.1">
</applet>
</center>
<hr>
<a href="ChatApplet.java">The source.</a>
</body>
```

```
</html>
```

A.1.2 ChatApplet.java

Este es el *applet* cliente de la aplicación. Se desarrolló con *Microsoft Visual J++ 1.0*:

```
//************************************************************
// ChatApplet.java: Applet
//
//************************************************************
import java.applet.*;
import java.awt.*;
import java.net.*;
import java.io.*;

//============================================================
// Main Class for applet ChatApplet
//
//============================================================
public class ChatApplet extends Applet implements Runnable
{
    // THREAD SUPPORT:
    //  m_ChatApplet    is the Thread object for the applet
    //--------------------------------------------------
    Thread   m_ChatApplet = null;

    // Applet data
    private TextField nameField;
    private TextArea chatZone;
    private TextField text;
    private Socket socket = null;
    private BufferedInputStream istream;
    private BufferedOutputStream ostream;

    private static int MAX_LEN = 500;

    // PARAMETER SUPPORT:
    //      Parameters allow an HTML author to pass
    // information to the applet; the HTML author specifies them
    // using the <PARAM> tag within the <APPLET> tag.  The
    // following variables are used to store the values of the
    // parameters.
    //--------------------------------------------------------

    // Members for applet parameters
    // <type>       <MemberVar>    = <Default Value>
    //--------------------------------------------------------
```

```
    private String m_direccion = "155.54.12.32";

    // Parameter names.  To change a name of a parameter,
    // you need only make a single change.  Simply modify the
    // value of the parameter string below.
    //--------------------------------------------------------
    private final String PARAM_direccion = "direccion";

    // ChatApplet Class Constructor
    //--------------------------------------------------------
    public ChatApplet()
    {
        // TODO: Add constructor code here
    }

    // APPLET INFO SUPPORT:
    //      The getAppletInfo() method returns a string
    // describing the applet's author, copyright date,
    // or miscellaneous information.
    //--------------------------------------------------------
    public String getAppletInfo()
    {
        return "Name: ChatApplet\r\n" +
               "Author: Diego\r\n" +
               "Created with Microsoft Visual J++ Version 1.0";
    }

    // PARAMETER SUPPORT
    //      The getParameterInfo() method returns an array
    // of strings describing the parameters understood by this applet.
    //
    // ChatApplet Parameter Information:
    //  { "Name", "Type", "Description" },
    //--------------------------------------------------------
    public String[][] getParameterInfo()
    {
        String[][] info =
        {
            { PARAM_direccion, "String","Parameter description" },
        };
        return info;
    }

    // The init() method is called by the AWT when an applet
    // is first loaded or reloaded.  Override this method to
    // perform whatever initialization your applet needs, such
    // as initializing data structures, loading images or
```

```
    // fonts, creating frame windows, setting the layout manager,
    // or adding UI components.
    //----------------------------------------------------------
    public void init()
    {
        // PARAMETER SUPPORT
        //      The following code retrieves the value
        // of each parameter specified with the <PARAM> tag and
        // stores it in a member variable.
        //------------------------------------------------
        String param;

        // direccion: Parameter description
        //------------------------------------------------
        param = getParameter(PARAM_direccion);
        if (param != null)
            m_direccion = param;

        // If you use a ResourceWizard-generated "control creator"
        // class to arrange controls in your applet, you may want
        // to call its CreateControls() method from within this
        // method. Remove the following call to resize() before
        // adding the call to CreateControls();
        // CreateControls() does its own resizing.
        //----------------------------------------------------------
        resize(600, 300);

        // TODO: Place additional initialization code here

        // Esto va a ser duro, pero...
        GridBagLayout gb = new GridBagLayout();
        GridBagConstraints c = new GridBagConstraints();
        setLayout(gb);

        // Poner Nombre: _________
        c.fill = GridBagConstraints.BOTH;
        c.weightx = 1.0;
        c.weighty = 1.0;
        Label l = new Label("Nombre:",Label.RIGHT);
        gb.setConstraints(l,c);
        add(l);

        c.gridwidth = GridBagConstraints.REMAINDER;
        nameField = new TextField("¡Pon tu nombre!");
        gb.setConstraints(nameField,c);
        add(nameField);
```

```
        // Ultimo de la línea: un TextArea que ocupa cuatro filas
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.gridheight = 8;
        chatZone = new TextArea();
        gb.setConstraints(chatZone,c);
        add(chatZone);

        // El texto que escribes
        c.gridheight = 1;
        text = new TextField();
        gb.setConstraints(text,c);
        add(text);

        //////////////////
        // Establecer la conexión
        try
        {
            socket = new Socket(m_direccion,9000);

            ostream = new BufferedOutputStream(
                    socket.getOutputStream());
            istream = new BufferedInputStream(
                    socket.getInputStream());
        } catch (Exception e)
        {
            showStatus("Error: "+e);
            chatZone.appendText(
                "Error estableciendo la comunicación. Sorry!\r\n");
        }

    }

    // Place additional applet clean up code here.  destroy()
    // is called when you applet is terminating and being unloaded.
    //-------------------------------------------------------------
    public void destroy()
    {
        // TODO: Place applet cleanup code here
    }

    //      The start() method is called when the page
    // containing the applet first appears on the screen. The
    // AppletWizard's initial implementation of this method
    // starts execution of the applet's thread.
    //-------------------------------------------------------------
    public void start()
    {
```

```
        if (m_ChatApplet == null)
        {
            m_ChatApplet = new Thread(this);
            m_ChatApplet.start();
        }
        // TODO: Place additional applet start code here
    }

    //      The stop() method is called when the page
    // containing the applet is no longer on the screen. The
    // AppletWizard's initial implementation of this method
    // stops execution of the applet's thread.
    //-------------------------------------------------------
    public void stop()
    {
        if (m_ChatApplet != null)
        {
            m_ChatApplet.stop();
            m_ChatApplet = null;
        }

        try
        {
            if (socket != null)
                socket.close();
            socket = null;
        }
        catch(Exception e)
        {}
    }

    // THREAD SUPPORT
    //      The run() method is called when the applet's
    // thread is started. If your applet performs any ongoing
    // activities without waiting for user input, the code for
    // implementing that behavior typically goes here. For
    // example, for an applet that performs animation, the run()
    // method controls the display of images.
    //-------------------------------------------------------
    public void run()
    {
        byte[] buffer = new byte[MAX_LEN];
        int bytesRead;
        String msg;

        nameField.selectAll();
```

```
        if (socket != null)
        {
            try
            {
                bytesRead = istream.read(buffer,0,MAX_LEN);
                while (bytesRead != -1)
                {
                    msg = new String(buffer,0,0,bytesRead);
                    chatZone.appendText(msg);

                    bytesRead = istream.read(buffer,0,MAX_LEN);
                }

                chatZone.appendText(
                    "\r\nąąą Conexión cortada por el servidor !!!\r\n");
                socket.close();
            } catch (Exception e)
            {
                showStatus("Error: " +e);
            }
        }
    }


    public boolean handleEvent(Event ev)
    {
        // Si en el cuadro de texto
        if (ev.target == text)
        {
            // Se ha pulsado una tecla
            if (ev.id == Event.KEY_PRESS)
                // Y es ENTER...
                if (ev.key == 0xa)
                {
                    sendToServer(nameField.getText() +
                        "> " + text.getText() + "\r\n");
                    text.setText("");
                    return true;
                }
        }

        return false;
    }


    private synchronized void sendToServer(String texto)
    {
```

```
        byte[] buffer = new byte[MAX_LEN];

        if (socket != null)
        {
            texto.getBytes(0,texto.length(),buffer,0);
            try {
                ostream.write(buffer,0,texto.length());
                ostream.flush();
            } catch (IOException e)
            {
                showStatus("Error mientras se escribía");
            }
        }
    }
}
```

A.1.3 ConnectionThread.java

La siguiente clase encapsula a la tarea que queda encargada de servir a un cliente en concreto:

```
import java.lang.Object;
import java.net.*;
import java.io.*;

/*
 *
 * ConnectionThread
 *
 */
public class ConnectionThread extends Thread implements Runnable
{
    private Socket socket = null;
    private Servidor parent = null;
    private BufferedInputStream in;
    private BufferedOutputStream out;

    ConnectionThread(Servidor p, Socket s)
    {
        socket = s;
        parent = p;
    }

    public BufferedOutputStream bos() { return out; };

    public void run()
    {
        int bytesRead;
```

```
        byte[] buffer = new byte[Servidor.MAX_LEN];

        try
        {
            in = new BufferedInputStream(socket.getInputStream());
            out = new BufferedOutputStream(socket.getOutputStream());

            bytesRead = in.read(buffer,0,Servidor.MAX_LEN);
            while (bytesRead != -1)
            {
                String msg = new String(buffer,0,0,bytesRead);

                parent.broadcast(msg);

                bytesRead = in.read(buffer,0,10);
            }

            System.out.println("Server part closed");
            socket.close();
            parent.unregister(this);
        } catch (Exception e)
        {
            System.err.println("Excepción: " + e);
        }
    }
}
```

A.1.4 Servidor.java

Servidor de la aplicación:

```
import java.lang.Object;
import java.net.*;
import java.io.*;

/*
 *
 * Servidor (buffered socket)
 *
 */
public class Servidor extends Object
{
    private static int MAX_CLIENTS = 2;
    public static int MAX_LEN = 500;
    private int Clients = 0;
    private ConnectionThread[] tareas;
```

```
    private Socket socket = null;

    public static void main(String[] args)
    {
        Servidor serv = new Servidor();
        serv.run();
    }

    public Servidor()
    {
        int i;

        tareas = new ConnectionThread[MAX_CLIENTS];
        for (i=0;i<MAX_CLIENTS;i++)
            tareas[i] = null;

        Clients = 0;
    }

    public void run()
    {
        PrintStream o = System.out;
        int j;

        try
        {
            o.println("Creando el socket del servidor");
            ServerSocket serverSocket = new ServerSocket(9000);

            while (true)
            {
                o.println("Performing accept");
                socket = serverSocket.accept();

                if (socket != null)
                {
                    if (Clients < MAX_CLIENTS)
                    {
                        // Create a thread to manage connection
                        o.println("Creating a server thread");
                        ConnectionThread t = new
                        ConnectionThread(this,socket);

                        for (j = 0;j<MAX_CLIENTS;j++)
                            if (tareas[j] == null)
                            {
                                tareas[j] = t;
```

```
                            break;
                        }

                    Clients++;
                    t.start();
                }
                else
                    socket.close();
            }
        }

    } catch (Exception e)
    {
        System.err.println("Excepción: " + e);
    }
}

public synchronized void broadcast(String msg)
{
    // Send broadcast to all clients
    int i;
    byte[] buffer = new byte[MAX_LEN];

    msg.getBytes(0,msg.length(),buffer,0);

    for(i=0;i< MAX_CLIENTS;i++)
    {
        if (tareas[i] != null)
        {
            try {
                tareas[i].bos().write(buffer,0,msg.length());
                tareas[i].bos().flush();
            } catch (IOException e)
            {
                System.err.println("Error mientras se escribąa");
            }
        }
    }
}

public synchronized void unregister(ConnectionThread thread)
{
    int i = 0;
    while (thread != tareas[i])
        i++;

    tareas[i] = null;
```

```
        Clients--;
    }
}
```

A.1.5 ChatClient.itcl

Este es un cliente escrito en [incr Tcl], Tcl Orientado a Objetos:

```
class ChatClient {

    private variable addr "127.0.0.1"
    private variable port 9000
    private variable sock ""

    constructor { a p } {
        set addr $a
        set port $p

        # abrir el socket
        set sock [socket $addr $port]

        fconfigure $sock -blocking 0
    }

    public method run {}
    private method sendToServer {}
    private method listenSocket {}

    destructor {
        if { $sock != "" } {
            close $sock
        }
    }
}

body ChatClient::run {} {

    # Crear los 'widgets' y ponerlos en la ventana
    labeledwidget .lw -labeltext "■Pon tu nombre!" -labelpos e
    entryfield .ef
    scrolledtext .st
    entryfield .fef -command [code $this sendToServer ]

    pack .lw .ef -fill both -expand yes
    pack .st -fill both -expand yes
```

```
    pack .fef -fill both -expand yes

    fileevent $sock readable [code $this listenSocket]
}



body ChatClient::listenSocket {} {
        set str [gets $sock]
        .st insert end "$str\n"
}


body ChatClient::sendToServer {} {
    set str "[.ef get]> [.fef get]"

    puts $sock $str
    flush $sock
    .fef clear
}

#ChatClient cc localhost 9000
#cc run
```

A.2 Aplicación de lista de correo con CGI

A continuación se listan las páginas HTML y programas CGI que pertenecen a la aplicación.

A.2.1 initdb.pl

Este programa crea la Bases de Datos que será utilizada por el programa a través de las llamadas del paquete `Win32::ODBC`:

```
use Win32::ODBC;

#
#   Configurar el Data Source para el Majordomo
#
$myDSN = "Base de Datos de Majordomo";
$driverType = "Microsoft Text Driver (*.txt; *.csv)";
$dir = ‘cd‘;
chop $dir;
$dir .= ’\db’;

#
```

```
#   Crear el DSN
#
print "Ad'adiendo el DSN \"$myDSN\"...";
if (Win32::ODBC::ConfigDSN( ODBC_ADD_DSN,
    $driverType,
    ("DSN=$myDSN", "Description=$myDSN",
        "DEFAULTDIR=$dir", "UID=", "PWD=")))
{
    print "OK!\n";
}
else
{
    print "Fallo\n" . Win32::ODBC::Error();
}
```

A.2.2 inittables.pl

Este programa inicializa las tablas que forman parte de base de datos necesaria para la aplicación. El borrado inicial con "`drop`" es preventivo y no necesario:

```
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

#$drh = DBI->install_driver("ODBC");

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $dbh->do("drop table List");
    $dbh->do("create table List (nombre char(40),email char(100),\
            adminpasswd char(20) )");

    $dbh->do("drop table Amos");
    $dbh->do("create table Amos (email char(100),lista char (100))" );

    $dbh->do("drop table Msgs");
    $dbh->do("create table Msgs (id char(10), sent char(1),lista char(40),\
        email char(100),fecha char(20),subject char(200),msg char(255))" );

}
else
{
    print "No connect";
}
```

A.2.3 header.html

Página parametrizada de encabezado, común a todas las páginas generadas por los programas CGI:

```
<HTML>
<HEAD>
   <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
   <META NAME="Author" CONTENT="diego">
   <META NAME="GENERATOR" CONTENT="Mozilla/4.02 [en] (Win95; I) [Netscape]">
   <TITLE><!--HEADERINFO--></TITLE>

   <!--JAVASCRIPT-->

</HEAD>
<BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000EE" VLINK="#551A8B" ALINK="#FF0

onLoad=
>
 
<CENTER><TABLE CELLSPACING=0 CELLPADDING=0 COLS=9 WIDTH="100%" >
<TR>
<TD>
<CENTER><FONT SIZE=+4>m</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>a</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>j</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>o</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>r</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>d</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>o</FONT></CENTER>
```

```
</TD>

<TD>
<CENTER><FONT SIZE=+4>m</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>o</FONT></CENTER>
</TD>
</TR>

<TR>
<TD>
<CENTER> <FONT SIZE=-2>(c)</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2>dsr </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2>1998 </FONT></CENTER>
</TD>
```

```
</TR>
</TABLE></CENTER>

<CENTER> </CENTER>

<CENTER> </CENTER>

<CENTER></CENTER>

<CENTER>
<HR WIDTH="100%"></CENTER>

<CENTER><FONT SIZE=+2><!--HEADERINFO--></FONT></CENTER>

<CENTER>
<HR WIDTH="100%"></CENTER>
```

A.2.4 footer.html

Parte equivalente a la anterior, pero de pie de página:

```
<p> </p>
<p> </p>
<DIV ALIGN=right>
<HR ALIGN=RIGHT WIDTH="12%"></DIV>

<DIV ALIGN=right> <A HREF="../newlist.html">Volver</A></DIV>

</BODY>
</HTML>
```

A.2.5 common.pm

Módulo común para todos los programas CGI de la aplicación:

```
# common.pl

$host = '127.0.0.1';

sub endHTML {
    open (FOOTER,"footer.html");
    print <FOOTER>;
    close(FOOTER);
}

sub printCabecera {
    my ($msg) = shift;
    my $cabecera;
```

```
    #
    #   Leer el contenido del fichero de cabecera
    #
    open (CAB,"header.html");
    $tmp = $/;
    undef $/;
    $cabecera = <CAB>;
    close (CAB);

    $cabecera =~ s/<!--HEADERINFO-->/$msg/g;
    print $cabecera;

    $/ = $tmp;
}

sub printError {
    my($msg) = shift;
    &printCabecera("ERROR");

    print "<blockquote>$msg</blockquote>";

    &endHTML;
}

1;
```

A.2.6 newlist.html

Página HTML que muestra el form para introducir una nueva lista de correo.

```
<HTML>
<HEAD>
   <META HTTP-EQUIV="Content-Type" CONTENT="text/html; charset=iso-8859-1">
   <META NAME="Author" CONTENT="diego">
   <META NAME="GENERATOR" CONTENT="Mozilla/4.02 [en] (Win95; I) [Netscape]">
   <TITLE>A&ntilde;adir Nueva Lista</TITLE>
</HEAD>
<BODY TEXT="#000000" BGCOLOR="#FFFFFF" LINK="#0000EE" VLINK="#551A8B" ALINK="#FF0
 
<CENTER><TABLE CELLSPACING=0 CELLPADDING=0 COLS=9 WIDTH="100%" >
<TR>
<TD>
<CENTER><FONT SIZE=+4>m</FONT></CENTER>
</TD>

<TD>
```

```
<CENTER><FONT SIZE=+4>a</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>j</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>o</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>r</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>d</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>o</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>m</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=+4>o</FONT></CENTER>
</TD>
</TR>

<TR>
<TD>
<CENTER> <FONT SIZE=-2>(c)</FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
```

```
</TD>

<TD>
<CENTER><FONT SIZE=-2>dsr </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2> </FONT></CENTER>
</TD>

<TD>
<CENTER><FONT SIZE=-2>1998 </FONT></CENTER>
</TD>
</TR>
</TABLE></CENTER>

<CENTER> </CENTER>

<CENTER> </CENTER>

<CENTER></CENTER>

<CENTER>
<HR WIDTH="100%"></CENTER>

<CENTER><FONT SIZE=+2>Nueva Lista de Correo</FONT></CENTER>

<CENTER>
<HR WIDTH="100%"></CENTER>

<BLOCKQUOTE>Al aceptar este formulario se crear&aacute; una lista de correo
con el t&iacute;tulo, e-mail y clave de administraci&oacute;n especificadas.
<BR> 
<BR> 

<P><FORM name="form1" action="http://127.0.0.1/cgi-bin/p?addlist.pl" method ="POS
<CENTER><TABLE WIDTH="75%" >
<TR>
<TD WIDTH="25%">Nombre de la lista</TD>
```

```
<TD><INPUT type="text" name="listname"></TD>
</TR>

<TR>
<TD>E-Mail para la lista</TD>

<TD><INPUT type="text" name="email"></TD>
</TR>

<TR>
<TD>Admin. Password</TD>

<TD><INPUT type="password" name="pass"></TD>
</TR>
</TABLE></CENTER>

<CENTER> </CENTER>

<CENTER> </CENTER>

<CENTER><INPUT type="SUBMIT" value="aceptar"></CENTER>
</FORM></BLOCKQUOTE>

</BODY>
</HTML>
```

A.2.7 addlist.pl

Programa CGI que procesa los datos del formulario anterior:

```
read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

@pairs = split(/&/,$buffer);

foreach $pair (@pairs)
{
    ($name,$value) = split(/=/,$pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
    $FORM{$name} = $value;
}

$email = $FORM{"email"};
$pass = $FORM{'pass'};
$nombre = $FORM{'listname'};
```

```
print "Content-type: text/html\n\n";

use common;

#
#   Comprobar que los datos son correctos
#
if ($email =~ /[ \t]/)
{
    &printError("El e-mail no puede llevar espacios");
}
else
{
    #
    #   Comprobar que la lista no existe
    #
    use DBI::W32ODBC;

    $myDSN = "Base de Datos de Majordomo";

    #$drh = DBI->install_driver("ODBC");
    $dbh = DBI->connect($myDSN,'','');
    if ($dbh)
    {
        $sth = $dbh->prepare("select * from List where email= \'$email\'");
        $sth->execute;
        if ($sth->fetchrow)
        {
            $sth->finish;
            $dbh->disconnect;
                &printError("ąąą La lista ya existe !!!");
        }
        else
        {
            $sth = $dbh->prepare("insert into List values \
                ( \'$nombre\' , \'$email\' , \'$pass\' )");
            $sth->execute;

            $sth = $dbh->prepare('select * from List');
            $sth->execute;

            &printCabecera("Lista Insertada");

            print "<blockquote><FONT SIZE=+1>Las siguientes \
                listas est&aacute;n mantenidas por ";
            print "este servidor:</FONT>";
            print "<BR> ";
```

```
            print "<UL>";

                while (($nombre,$email) = $sth->fetchrow) {
                print "<LI><FONT SIZE=+1>$nombre (<B>e-mail:\
                    </B> $email)</FONT></LI>";
            }

            print "</UL></blockquote>";

            $sth->finish;
            $dbh->disconnect;

            &endHTML;
        }
    }
    else
    {
        &printError("Error en la base de datos");
    }
}
```

A.2.8 search.pl

Muestra el formulario de búsqueda como una página HTML:

```
print "Content-type: text/html\n\n";

use common;
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("select * from List");
    $sth->execute;

    &printCabecera("B\&uacute;squedas");

    print "<P><FORM name=\"form1\"";
    print "action=\"http://$host/cgi-bin/p?dosearch.pl\"";
    print "method =\"POST\">";

    print "<CENTER><TABLE WIDTH=\"100%\" >";
    print "<TR>";
    print "<TD WIDTH=\"25%\">Lista</TD>";
```

```
print "<TD><select name=\"listname\">";

#
#   Insertar las distintas listas
#
while (($nombre,$email) = $sth->fetchrow)
{
    print "<option>$nombre";
}

print "</select>";

print "</TD>";
print "</TR>";

print "<TR>";
print "<td>Fecha Desde</td>";
print "<TD><input type=\"text\" size=20 name=\"fechadesde\"></TD>";
print "</TR>";

print "<TR>";
print "<td>Fecha Hasta</td>";
print "<TD><input type=\"text\" size=20 name=\"fechahasta\"></TD>";
print "</TR>";

print "<TR>";
print "<td>BUSCAR</td>";
print "<TD><input type=\"text\" size=50 name=\"buscar\"></TD>";
print "</TR>";

print "<TR>";
print "<td></td>";
print "<TD><input type=\"radio\" name=\"tipo\"\
         value=\"sub\" checked>\ Subcadenas</TD>";
print "</TR>";

print "<TR>";
print "<td></td>";
print "<TD><input type=\"radio\" name=\"tipo\"\
         value=\"regex\">\ Expresi\&oacute;n Regular</TD>";
print "</TR>";

print "<TR>";
print "<td>Buscar en:</td>";
print "<TD>";
print "<input type=\"checkbox\" name=\"subject\"\
```

```
            value=\"subject\" checked>\ Subject\ ";
    print "<input type=\"checkbox\" name=\"texto\"\
            value=\"texto\">\ Texto\ ";
    print "</td>";
    print "</TR>";


    print "</TABLE>";

    print "<input type=\"submit\" value=\"Buscar !\">";
    print "</form>";
    print "</center>";

    $sth->finish;
    $dbh->disconnect;

    &endHTML;
}
else
{
    &printError("Error en la base de datos");
}
```

A.2.9 dosearch.pl

Realiza la búsqueda requerida por el anterior formulario. Esto puede ser considerado como un *engine* sencillo de búsqueda:

```
#!/usr/bin/perl

read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

@pairs = split(/&/,$buffer);

foreach $pair (@pairs)
{
    ($name,$value) = split(/=/,$pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
    $FORM{$name} = $value;
}

print "Content-type: text/html\r\n\r\n";

use common;
```

```
$lista = $FORM{'listname'};
$buscar = $FORM{'buscar'};
$tipo = $FORM{'tipo'};
$subject = defined($FORM{'subject'});
$texto = defined($FORM{'texto'});

&printCabecera("Resultados de la b\&uacute;squeda");

#
#   Buscar ahora en todos los mensajes
#
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("select * from Msgs where\
            lista = \'$lista\'");
    $sth->execute;

    #
    #   El siguiente filtrado lo hacemos nosotros
    #
    while (($dbId,$dbSent,$dbLista,$dbEmail,$dbFecha,$dbSubject,$dbMsg)
            = $sth->fetchrow)
    {
        $testStr = "";
        $testStr .= $dbSubject  if $subject;
        $testStr .= $dbMsg if $texto;

        if ($testStr =~ m|$buscar|ig)
        {
            #
            #   Esta entrada es válida
            #
            print "<BR><P><TABLE WIDTH=\"100%\">";
            print "<tr><td width=\"80%\">";
            print "<b>From:\ </b>$dbEmail<br>";
            print "<b>At:\ </b>$dbFecha<br>";
            print "<b>Subject: </b>$dbSubject<br>";
            print "</td>";
            print "<td>";
            print "<form name=\"miform\">";
            print "<input type=\"button\" value=\"Ver\" onClick=\
```

```
                    \"window.open(\'http://$host/cgi-bin/p?show.pl+$dbId\'\
                    ,\'newWin\',\'toolbar=no,directories=no\')\">";
            print "</form>";
            print "</td>";
            print "</tr></table>";
        }
    }

    $sth->finish;
    $dbh->disconnect;

    &endHTML;

}
else
{
    &printError("Error en la base de datos");
}
```

A.2.10 sendmsg.pl

Muestra el formulario que permite enviar un mensaje:

```
print "Content-type: text/html\n\n";

use common;
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("select * from List");
    $sth->execute;

    open(CAB,"header.html");
    $tmp = $/;
    undef $/;
    $cabecera = <CAB>;
    $cabecera =~ s/<!--HEADERINFO-->/Enviar Mensaje/g;
    $cabecera =~ s/onLoad=/onLoad=\"setdate()\"/gi;

    $javaScriptProgram="<SCRIPT languaje=\"JavaScript\">\
                    <!-- \
```

```
                function setdate() {\
                        var now = new Date();\
                        var dia = now.getDate();\
                        var mes = now.getMonth() + 1;\
                        var anno = now.getYear(); \
                        \
                        var texto = dia + \"/\" + mes + \"/\" + anno; \
                        \
                        document.form1.fecha.value = texto; \
                }\
                // --> \
                </SCRIPT> ";



$cabecera =~ s|<!--JAVASCRIPT-->|$javaScriptProgram|g;
print $cabecera;

undef $cabecera;
close(CAB);
$/ = $tmp;

print "<P><FORM name=\"form1\"";

#
#   Temporary
#

print "action=\"http://$host/cgi-bin/t?newmsg.tcl\"";
print "method =\"POST\">";

print "<CENTER><TABLE WIDTH=\"100%\" >";
print "<TR>";
print "<TD WIDTH=\"10%\">Lista</TD>";

print "<TD><select name=\"listname\">";

#
#   Insertar las distintas listas
#
while (($nombre,$email) = $sth->fetchrow)
{
    print "<option>$nombre";
}

print "</select>";
```

```
    print "</TD>";
    print "</TR>";

    print "<TR>";
    print "<td>Fecha</td>";
    print "<TD><input type=\"text\" size=20 name=\"fecha\"></TD>";
    print "</TR>";

    print "<TR>";
    print "<td>Remite</td>";
    print "<TD><input type=\"text\" size=40 name=\"remite\"></TD>";
    print "</TR>";

    print "<TR>";
    print "<TD>Subject</TD>";

    print "<TD><input type=\"text\" size=40 name=\"subject\"></TD>";
    print "</TR>";

    print "<TR>";
    print "<TD>Msg:</TD>";
    print "<TD>";
    print "<textarea name=\"msg\" rows=10 cols=50></textarea>";
    print "</TD>";
    print "</TR>";

    print "</TABLE>";

    print "<input type=\"submit\" value=\"Enviar\">";
    print "</form>";
    print "</center>";

    $sth->finish;
    $dbh->disconnect;

    &endHTML;
}
else
{
    &printError("Error en la base de datos");
}
```

A.2.11 newmsg.pl

Programa que procesa los datos del formulario anterior y que los almacena en la base de datos:

```
#!/usr/bin/perl

read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

@pairs = split(/&/,$buffer);

foreach $pair (@pairs)
{
    ($name,$value) = split(/=/,$pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
    $FORM{$name} = $value;
}

print "Content-type: text/html\r\n\r\n";

use common;

$lista = $FORM{'listname'};
$email = $FORM{'remite'};
$fecha = $FORM{'fecha'};
$subject = $FORM{'subject'};
$msg = $FORM{'msg'};
$msg =~ s/\n/_/g;

&printCabecera('Inserci&oacute;n de un mensaje');

#
#   Insertar ahora el mensaje
#
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    #
    #   Actualizar el contador
    #
    open(ID,"contador");
    chop($id = <ID>);
    close(ID);


    $sth = $dbh->prepare("insert into Msgs values\
         (\'$id\',\'N\',\'$lista\',\'$email\',\'$fecha\',\
```

```
        \'$subject\',\'$msg\')");
    $sth->execute;

    print "<CENTER>Su mensaje ha sido enviado a todos \
        los componentes de la lista $lista.</CENTER>";

    $sth->finish;

    open(ID,">contador");
    ++$id;
    print ID "$id\r\n";
    close(ID);

#    $sth = $dbh->prepare("update Cont set Id = Id + 1 where Id = \'$id\'");
#    $sth->execute;

    $dbh->disconnect;

    &endHTML;

}
else
{
    &printError("Error en la base de datos");
}
```

A.2.12 newmsg.tcl

Este es el mismo programa que el anterior, pero en este caso, implementado en **Tcl**. Así sirve de comparativa entre estos dos lenguajes:

```
#!/usr/bin/tclsh  # -*- Tcl -*-

set serverData [read stdin $env(CONTENT_LENGTH)]
#set serverData "yo=tu+other%64&el=ella"

puts "Content-type: text/html\r\n\r\n"

# Decode encoded QUERY data
proc decode {str} {
    global FORM
    foreach pair [split $str &] {
        # Proteger los caracteres especiales de Tcl (de tclhttpd 2.1.2)
        regsub -all {[][\\$]} $pair {\\&} pair
```

```
        set nameandval [split $pair =]
        set name [lindex $nameandval 0]
        set val [lindex $nameandval 1]

        # Cambiar + por { } a $val
        regsub -all {\+} $val { } val

        # Sustituir las cadenas %xx por su valor ASCII
        regsub -all {%([a-zA-z0-9][a-zA-Z0-9])} $val {[format %c 0x\1]} val

        # Actualizar FORM y producir las sustituciones de [format]
        set FORM($name) [subst $val]
    }
}

proc printCabecera {msg} {
    set head [open header.html]
    set file [read $head]

    # La siguiente expresićn es un poco curiosa...
    regsub -all {[\\&]} $msg {\\&} msg
    regsub -all {<!--HEADERINFO-->} $file "$msg" file

    puts $file
    close $head
}

proc printPie {} {
    set foot [open footer.html]
    set file [read $foot]
    puts $file
    close $foot
}

decode $serverData
printCabecera {Inserci&oacute;n de un mensaje}

load /info/tcl/tclodbc.zip/tclodbc

set lista $FORM(listname);
set email $FORM(remite);
set fecha $FORM(fecha);
set subject $FORM(subject);
set msg $FORM(msg);
regsub -all {\n} $msg _ msg

# Conectar con la base de datos
```

```
database connect db {Base de Datos de Majordomo}

# Inicializar contador
set counter [open "contador"]
set id [gets $counter]
close $counter

db statement insert "insert into Msgs values (’$id’, ’N’, ’$lista’,\
                    ’$email’, ’$fecha’, ’$subject’, ’$msg’)"
insert execute

incr id

set counter [open "contador" w]
puts $counter $id
close $counter

db disconnect

puts  "<CENTER>Su mensaje ha sido enviado a todos \
         los componentes de la lista $lista.</CENTER>"

printPie
```

A.2.13 subscribe.pl

Prepara el formulario que permite suscribirse a una lista de correo:

```
print "Content-type: text/html\n\n";

use common;
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,’’,’’);
if ($dbh)
{
    $sth = $dbh->prepare("select * from List");
    $sth->execute;

    &printCabecera("Suscripci\&oacute;n a una lista");

    print "<P><FORM name=\"form1\"";
    print "action=\"http://$host/cgi-bin/p?newsub.pl\"";
    print "method =\"POST\">";
```

```
    print "<CENTER><TABLE WIDTH=\"75%\" >";
    print "<TR>";
    print "<TD WIDTH=\"25%\">Lista</TD>";

    print "<TD><select name=\"listname\">";

    #
    #   Insertar las distintas listas
    #
    while (($nombre,$email) = $sth->fetchrow)
    {
        print "<option>$nombre";
    }

    print "</select>";

    print "</TD>";
    print "</TR>";

    print "<TR>";
    print "<td>E-Mail</td>";
    print "<TD><input type=\"text\" size=40 name=\"email\"></TD>";
    print "</TR>";

    print "</TABLE>";

    print "<input type=\"submit\" value=\"Suscribirse\">";
    print "</form>";
    print "</center>";

    $sth->finish;
    $dbh->disconnect;

    &endHTML;
}
else
{
    &printError("Error en la base de datos");
}
```

A.2.14 newsub.pl

Procesa el anterior formulario, actualizando la base de datos:

```
#!/usr/bin/perl
```

```
read (STDIN, $buffer, $ENV{'CONTENT_LENGTH'});

@pairs = split(/&/,$buffer);

foreach $pair (@pairs)
{
    ($name,$value) = split(/=/,$pair);
    $value =~ tr/+/ /;
    $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",hex($1))/eg;
    $FORM{$name} = $value;
}

print "Content-type: text/html\r\n\r\n";

use common;

$lista = $FORM{'listname'};
$email = $FORM{'email'};

&printCabecera('Suscripci&oacute;n a una lista');

#
#   Insertar ahora la suscripcićn
#
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("insert into Amos values (\'$email\',\'$lista\')");
    $sth->execute;

    print "<CENTER>Ha quedado suscrito a la lista $lista.</CENTER>";

    $sth->finish;
    $dbh->disconnect;

    &endHTML;

}
else
{
    &printError("Error en la base de datos");
}
```

A.2.15 response.pl

Muestra el formulario que permite responder a un mensaje anterior:

```
print "Content-type: text/html\n\n";

use common;

#
#   Leer de QUERY_STRING y eliminar el nombre del programa.
#   Esto sólo lo tenemos que hacer en Win95
#
($dummy,$id) = split(/\+/,$ENV{'QUERY_STRING'});

use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";

$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("select * from Msgs where\
            Id = \'$id\'");
    $sth->execute;

    #
    #   Recuperar el mensaje
    #
    ($dbId,$dbSent,$dbLista,$dbEmail,$dbFecha,$dbSubject,$dbMsg)
        = $sth->fetchrow;

    open(CAB,"header.html");
    $tmp = $/;
    undef $/;
    $cabecera = <CAB>;
    $cabecera =~ s/<!--HEADERINFO-->/Responder a Mensaje/g;
    $cabecera =~ s/onLoad=/onLoad=\"setdate()\"/gi;

    $javaScriptProgram="<SCRIPT languaje=\"JavaScript\">\
                    <!-- \
                    function setdate() {\
                            var now = new Date();\
                            var dia = now.getDate();\
                            var mes = now.getMonth() + 1;\
```

```
                        var anno = now.getYear(); \
                        \
                        var texto = dia + \"/\" + mes + \"/\" + anno; \
                        \
                        document.form1.fecha.value = texto; \
                }\
                // --> \
                </SCRIPT> ";


$cabecera =~ s|<!--JAVASCRIPT-->|$javaScriptProgram|g;
print $cabecera;

undef $cabecera;
close(CAB);
$/ = $tmp;

print "<P><FORM name=\"form1\"";
print "action=\"http://$host/cgi-bin/p?newmsg.pl\"";
print "method =\"POST\">";

print "<CENTER><TABLE WIDTH=\"100%\" >";
print "<TR>";
print "<TD WIDTH=\"10%\">Lista</TD>";
print "<TD><input type=\"text\" size=20 name=\"listname\"\
       value=\"$dbLista\"></Td>";
print "</TR>";

print "<TR>";
print "<td>Fecha</td>";
print "<TD><input type=\"text\" size=20 name=\"fecha\"></TD>";
print "</TR>";

print "<TR>";
print "<td>Remite</td>";
print "<TD><input type=\"text\" size=40 name=\"remite\"></TD>";
print "</TR>";

print "<TR>";
print "<TD>Subject</TD>";

print "<TD><input type=\"text\" size=40 name=\"subject\"\
       value=\"Re: $dbSubject\"></TD>";
print "</TR>";

print "<TR>";
```

```
    print "<TD>Msg:</TD>";
    print "<TD>";
    print "<textarea name=\"msg\" rows=10 cols=50></textarea>";
    print "</TD>";
    print "</TR>";

    print "</TABLE>";

    print "<input type=\"submit\" value=\"Enviar\">";
    print "</form>";
    print "</center>";

    $sth->finish;
    $dbh->disconnect;

    &endHTML;
}
else
{
    &printError("Error en la base de datos");
}
```

A.2.16 show.pl

Muestra un mensje y permite responderlo:

```
print "Content-type: text/html\r\n\r\n";

use common;

#
#   Leer de QUERY_STRING y eliminar el nombre del programa.
#   Esto sćlo lo tenemos que hacer en Win95
#
($dummy,$id) = split(/\+/,$ENV{'QUERY_STRING'});

&printCabecera("Mensaje");

#
#   Buscar el mensaje en cuestićn
#
use DBI::W32ODBC;

$myDSN = "Base de Datos de Majordomo";
```

```
$dbh = DBI->connect($myDSN,'','');
if ($dbh)
{
    $sth = $dbh->prepare("select * from Msgs where\
            Id = \'$id\'");
    $sth->execute;

    #
    #   Recuperar el mensaje
    #
    ($dbId,$dbSent,$dbLista,$dbEmail,$dbFecha,$dbSubject,$dbMsg)
        = $sth->fetchrow;

    print "<P><FORM name=\"form1\">";

    print "<CENTER><TABLE WIDTH=\"100%\" >";
    print "<TR>";
    print "<TD WIDTH=\"10%\"><b>Lista</b></TD>";

    print "<TD>$dbLista";
    print "</TD>";
    print "</TR>";

    print "<TR>";
    print "<td><b>Fecha</b></td>";
    print "<TD>$dbFecha</TD>";
    print "</TR>";

    print "<TR>";
    print "<td><b>Remite</b></td>";
    print "<TD>$dbEmail</TD>";
    print "</TR>";

    print "<TR>";
    print "<TD><b>Subject</b></TD>";
    print "<TD>$dbSubject</TD>";
    print "</TR>";

    $dbMsg =~ s/_/\n/g;

    print "<TR>";
    print "<TD><b>Msg:</b></TD>";
    print "<TD>";
    print "<textarea name=\"msg\" rows=10 cols=50>$dbMsg</textarea>";
    print "</TD>";
    print "</TR>";
```

```
    print "</TABLE>";

    print "<input type=\"button\" value=\"Cerrar\"\
        onClick=\"window.close()\">";

    print '   ';

    print "<input type=\"button\" value=\"Responder\"\
        onClick=\"window.open(\'http://$host/cgi-bin/p?response.pl+$id\',\
        \'otherNewWin\',\'toolbar=no,directories=no\')\">";

    print "</form>";
    print "</center>";


    $sth->finish;
    $dbh->disconnect;

    &endHTML;

}
else
{
    &printError("Error en la base de datos");
}
```

A.3 Aplicación de lista de correo con RMI

Las modificaciones a la aplicación basada en CGI se vieron en la sección 3.4. Aquí se presentan todos los listados.

A.3.1 RemoteCollection.java

Interfaz remoto definido por la aplicación:

```
package majordomo;

import java.util.*;
import java.rmi.*;

public interface RemoteCollection extends java.rmi.Remote
{
    // Inserción y eliminación
    int add(Object o) throws RemoteException;
```

```
    int remove() throws RemoteException;


    // Consulta secuencial
    int first() throws RemoteException;
    Object nextElement() throws RemoteException;
    void close() throws RemoteException;

    // Otros...
}
```

A.3.2 Lista.java

Los objetos de la clase "`Lista`" representa listas de correo:

```
package majordomo;

import java.io.Serializable;

public class Lista implements Serializable
{
    private String nombre;
    private String email;
    private String adminpasswd;

    public Lista(String n,String e,String pass)
    {
        nombre = n;
        email = e;
        adminpasswd = pass;
    }

    public String getNombre() { return nombre; }
    public String getEmail() { return email; }
    public String getAdminpasswd() { return adminpasswd; }

    // Otros mĆtodos aquą...
}
```

A.3.3 Msg.java

Mensajes que se incluyen dentro de una lista de correo:

```
package majordomo;

import java.io.Serializable;

public class Msg implements Serializable
```

```
{
    private String Id;
    private String sent;
    private String lista;
    private String email;
    private String fecha;
    private String subject;
    private String msg;

    public Msg(String i,String s,String list,
        String e,String f,String sub, String msg)
    {
        Id = i;
        sent = s;
        email = e;
        lista = list;
        fecha = f;
        subject = sub;
        this.msg = msg;
    }

    public String getId() { return Id; }
    public String getEmail() { return email; }
    public String getSent() { return sent; }
    public String getLista() { return lista; }
    public String getFecha() { return fecha; }
    public String getSubject() { return subject; }
    public String getMsg() { return msg; }

    // Otros mĆtodos aquą...
}
```

A.3.4 GenericRemoteCollection.java

Clase base de las colecciones genéricas:

```
package majordomo;

import majordomo.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
import java.rmi.*;
import java.sql.*;

public abstract class GenericRemoteCollection
    extends java.rmi.server.UnicastRemoteObject
    implements majordomo.RemoteCollection
```

```
{
    protected Connection con;
    protected Statement st;
    protected ResultSet rs;

    // Las subclases pueden indicar la tabla que abstraen
    protected abstract String tableName();

    public GenericRemoteCollection(String dataSource)
    throws RemoteException
    {
        super();

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        } catch (ClassNotFoundException e)
        {
            System.out.println("No se encuentra la clase JdbcOdbcDriver.");
            return;
        }

        try {
            con = DriverManager.getConnection("jdbc:odbc:"+dataSource,"","");
        } catch (java.sql.SQLException e)
        {
            System.err.println("Error: " +e );
            e.printStackTrace();
        }

        st = null;
    }

    protected void finalize() throws Throwable
    {
        try {
            con.close();
            close();
        } catch (java.sql.SQLException ignored)
        {}
    }

    public abstract int remove() throws RemoteException;
    public abstract int add(Object o) throws RemoteException;

    public int first() throws RemoteException
    {
        try {
```

```
            if (st != null)
                st.close();

            st = con.createStatement();
            rs = st.executeQuery("select * from " + tableName());
        } catch (java.sql.SQLException e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }

        return 0;
    }

    public abstract Object nextElement() throws RemoteException;

    public void close() throws RemoteException
    {
        try {
            if (st != null)
            {
                rs.close();
                st.close();
                st = null;
                rs = null;
            }
        } catch (SQLException e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }
    }
}
```

A.3.5 ListaCollection.java

Implementación de una colección remota de `Lista`s:

```
package majordomo;

import java.rmi.*;
import majordomo.*;
import java.sql.*;

public class ListaCollection extends GenericRemoteCollection
{
    public ListaCollection(String dataSource) throws RemoteException
```

```
    {
        super(dataSource);
    }

    protected String tableName() { return "List"; }

    public int remove() throws RemoteException
    {
        // No Implementado !!!
        return 0;
    }

    public int add(Object o) throws RemoteException
    {
        // No Implementado !!!
        return 0;
    }

    public Object nextElement() throws RemoteException
    {
        if (rs == null)
            return null;

        try {
            if (rs.next())
            {
                // Construir un nuevo objeto Lista y enviarlo
                return new Lista(   rs.getString("nombre"),
                                    rs.getString("email"),
                                    rs.getString("adminpasswd"));
            }
        } catch (SQLException e)
        {
            System.err.println("Error: " +e);
            e.printStackTrace();
        }
        return null;
    }

}
```

A.3.6 MsgCollection.java

Implementación de una colección remota de `Msg`s:

```
package majordomo;
```

```
import java.sql.*;
import java.rmi.*;
import majordomo.*;

public class MsgCollection extends GenericRemoteCollection
{
    public MsgCollection(String dataSource) throws RemoteException
    {
        super(dataSource);
    }

    protected String tableName() { return "Msgs"; }

    public int remove() throws RemoteException
    {
        // No Implementado !!!
        return 0;
    }

    public int add(Object o) throws RemoteException
    {
        Msg tmpMsg;
        try {
            if (rs != null)
            {
                rs.close();
                st.close();
                rs = null;
            }

            tmpMsg = (Msg)o;

            st = con.createStatement();
            st.executeUpdate("insert into Msgs values ('"+
                                        tmpMsg.getId() + "','" +
                                        "N" + "','" +
                                        tmpMsg.getLista() + "','"+
                                        tmpMsg.getEmail() + "','"+
                                        tmpMsg.getFecha() + "','"+
                                        tmpMsg.getSubject() + "','"+
                                        tmpMsg.getMsg() + "')");

            st.close();
            st = null;

        } catch (SQLException e)
        {
```

```
            System.err.println("Error: "+ e);
            e.printStackTrace();
        }
        return 0;
    }

    public Object nextElement() throws RemoteException
    {
        if (rs == null)
            return null;
        try {
            if (rs.next())
            {
                // Construir un nuevo objeto Lista y enviarlo
                return new Msg( rs.getString("id"),
                                rs.getString("sent"),
                                rs.getString("lista"),
                                rs.getString("email"),
                                rs.getString("fecha"),
                                rs.getString("subject"),
                                rs.getString("msg"));
            }
        } catch (SQLException e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }

        return null;
    }

}
```

A.3.7 sendmsg.html

Página HTML que incluye al *applet* de envío de un mensaje:

```
<html>

<head>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
<meta name="Author" content="diego">
<meta name="GENERATOR" content="Microsoft FrontPage Express 2.0">
<title>Enviar Mensaje</title>
</head>
```

```
<body bgcolor="#FFFFFF" text="#000000" link="#0000EE"
vlink="#551A8B" alink="#FF0000">

<p align="center">  </p>
<div align="center"><center>

<table border="0" cellpadding="0" cellspacing="0" width="100%"
cols="9">
    <tr>
        <td><font size="7">m</font> </td>
        <td><font size="7">a</font> </td>
        <td><font size="7">j</font> </td>
        <td><font size="7">o</font> </td>
        <td><font size="7">r</font> </td>
        <td><font size="7">d</font> </td>
        <td><font size="7">o</font> </td>
        <td><font size="7">m</font> </td>
        <td><font size="7">o</font> </td>
    </tr>
    <tr>
        <td> <font size="1">(c)</font> </td>
        <td><font size="1"> </font> </td>
        <td><font size="1"> </font> </td>
        <td><font size="1"> </font> </td>
        <td><font size="1">dsr </font> </td>
        <td><font size="1"> </font> </td>
        <td><font size="1"> </font> </td>
        <td><font size="1"> </font> </td>
        <td><font size="1">1998 </font> </td>
    </tr>
</table>
</center></div>

<p align="center">    </p>

<hr>

<p align="center"><font size="5">Enviar Mensaje</font></p>

<hr>

<p><center><applet code="SendMsgApplet.class" align="baseline"
width="430" height="500"></applet></center></p>

<p> </p>

<p> </p>
```

```
<hr align="right" width="12%">

<p align="right"> <a href="../newlist.html">Volver</a> </p>
</body>
</html>
```

A.3.8 SendMsgApplet.java

Applet de envío de mensaje:

```
import java.awt.*;
import java.applet.*;
import majordomo.*;
import java.rmi.*;
import java.util.Date;


public class SendMsgApplet extends Applet {
    void button1_Clicked(Event event) {
        try {
            RemoteCollection Mensajes = (RemoteCollection)Naming.lookup(
                    "//127.0.0.1/MsgCollection");

            Mensajes.first();

            String mmm = textArea1.getText();

            Msg m = new Msg(new Long(System.currentTimeMillis()).toString(),
                    "N",
                    choice1.getSelectedItem(),
                    textField2.getText(),
                    textField1.getText(),
                    textField3.getText(),
                    mmm.replace('\n','_')
                    );
            Mensajes.add( m );

        } catch (Exception e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }


    }
```

```
public void init() {
    super.init();

    //{{INIT_CONTROLS
    setLayout(null);
    addNotify();
    resize(422,473);
    setBackground(new Color(16777215));
    choice1 = new java.awt.Choice();
    initChoice();
    add(choice1);
    choice1.reshape(112,53,119,100);
    choice1.setBackground(new Color(16777215));
    label1 = new java.awt.Label("Lista");
    label1.reshape(28,53,77,22);
    label1.setFont(new Font("Dialog", Font.PLAIN, 16));
    add(label1);
    textField1 = new java.awt.TextField();
    textField1.reshape(112,90,119,23);
    Date now = new Date();
    textField1.setText(
        now.getDate()+"/"
        + (new Integer(now.getMonth()+1).toString())+"/"
        +now.getYear());
    add(textField1);
    label2 = new java.awt.Label("Fecha");
    label2.reshape(28,90,70,15);
    label2.setFont(new Font("Dialog", Font.PLAIN, 16));
    add(label2);
    textField2 = new java.awt.TextField();
    textField2.reshape(112,128,280,23);
    add(textField2);
    label3 = new java.awt.Label("Remite");
    label3.reshape(28,128,70,15);
    label3.setFont(new Font("Dialog", Font.PLAIN, 16));
    add(label3);
    textField3 = new java.awt.TextField();
    textField3.reshape(112,165,280,23);
    add(textField3);
    label4 = new java.awt.Label("Subject");
    label4.reshape(28,165,70,15);
    label4.setFont(new Font("Dialog", Font.PLAIN, 16));
    add(label4);
    textArea1 = new java.awt.TextArea();
    textArea1.reshape(28,225,367,161);
```

```
        add(textArea1);
        button1 = new java.awt.Button("Enviar");
        button1.reshape(168,428,88,25);
        add(button1);
        label5 = new java.awt.Label("Mensaje");
        label5.reshape(28,203,70,15);
        label5.setFont(new Font("Dialog", Font.PLAIN, 16));
        add(label5);
        //}}
    }

    public boolean handleEvent(Event event) {
        if (event.target == button1
            && event.id == Event.ACTION_EVENT) {
            button1_Clicked(event);
        }
        return super.handleEvent(event);
    }

    private void initChoice()
    {
        try {
            RemoteCollection listas = (RemoteCollection)Naming.lookup(
                    "//127.0.0.1/ListaCollection");

            listas.first();
            Lista l;
            while ((l = (Lista)listas.nextElement()) != null)
            {
                choice1.addItem(l.getNombre());
            }
            listas.close();
        } catch (Exception e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }
    }

    //{{DECLARE_CONTROLS
    java.awt.Choice choice1;
    java.awt.Label label1;
    java.awt.TextField textField1;
    java.awt.Label label2;
    java.awt.TextField textField2;
    java.awt.Label label3;
    java.awt.TextField textField3;
```

```
    java.awt.Label label4;
    java.awt.TextArea textArea1;
    java.awt.Button button1;
    java.awt.Label label5;
    //}}
}
```

A.3.9 Server.java

Servidor RMI de los objetos colección remota:

```
package majordomo;

import majordomo.*;
import java.rmi.*;
import java.rmi.server.*;


public class Server
{
    public static void main(String args[])
    {
        System.setSecurityManager(new RMISecurityManager());

        try {
            ListaCollection lc =
        new ListaCollection("Base de Datos de Majordomo");
            Naming.rebind("//127.0.0.1/ListaCollection",lc);

            MsgCollection mc =
        new MsgCollection("Base de Datos de Majordomo");
            Naming.rebind("//127.0.0.1/MsgCollection",mc);
        } catch (Exception e)
        {
            System.err.println("Error: "+e);
            e.printStackTrace();
        }
    }
}
```

Apéndice B

Una introducción personal a los lenguajes de *script*

La experiencia obtenida con la realización de este proyecto ha demostrado que una documentación que de una forma clara y rápida muestre las características principales de cualquier herramienta, lenguaje o tecnología es mucho más valiosa en la etapa de iniciación que los grandes manuales de referencia.

Durante el desarrollo de este proyecto hemos tenido la oportunidad de trabajar con lenguajes de programación de *script* en los que hemos escrito programas de demostración y prueba. Hemos hecho referencia a ellos a lo largo de este proyecto, pero no hemos dado ninguna guía de iniciación. Esto es lo que pretendemos, brevemente, claro está, en este capítulo. Se ofrecen además referencias adicionales que ayudarán a obtener un conocimiento más profundo sobre cada uno de los lenguajes. Los lenguajes que tratamos son **Perl**, **Tcl/Tk** y **Python**.

B.1 PERL

Perl ([WS92], [Gar98]), *Practical Extraction and Report Languaje* ó *Pathologically Eclectic Rubbish Lister*, fue creado por Larry Wall a mediados de los 80. Fue ideado inicialmente como un lenguaje de generación de resúmenes de *logs* en formato texto. Actualmente, en su versión 5, ya posee características avanzadas como módulos y clases y una librería muy grande de paquetes adicionales.

La principal crítica que recibe el lenguaje es su poca legibilidad. De echo, el *slogan* del lenguaje es "*There's more than one way to do it*" (Hay más de un modo de hacerlo). Parte de esta fama la recibe, por ejemplo, de que existen una serie de operadores que actúan "mágicamente" sobre la variable especial "`$_`", que guarda la línea actual de la entrada.

Las variables que se pueden definir en Perl son de tres tipos: *escalares*, como cadenas de caracteres y enteros; *listas*; ó *tablas de asociación*. Las primeras se forman anteponiendo el carácter "`$`" a su nombre; las listas anteponiendo el "`@`"; y las tablas de asociación, anteponiendo "`%`". Así, uno puede escribir:

```
$escalar = 2;
$cadena = "hola";
@lista = (1, 2, 3);
%asoc = ('nombre' => 'Diego');
```

Las variables pueden ser impresas con el comando "`print`":

```
print $escalar,"\t",$cadena;        => 2    hola
print @lista;                       => 123
```

donde la parte derecha de la flecha "`=>`" indica la salida. Nótese cómo en la segunda línea, los elementos que forman parte de la lista no se separan. Para conseguirlo se puede utilizar la función *built-in* "`join`":

```
print join(",",@lista);             => 1,2,3
```

Hay un gran conjunto de funciones disponibles, como vimos en el capítulo dedicado a los sockets, hay otras que dan soporte a la programación con esta tecnología, etc.

La regla de anteponer un `$` se sigue también para los elementos de una lista y de una tabla de asociación. Por ejemplo, para acceder al primer elemento de la lista `@lista` y al campo "`nombre`" de `%asoc`, se debe escribir:

```
$elemento = $lista[0];
$nombre = $asoc{'nombre'};
```

con lo que `$elemento` queda con el valor "`1`", y la variable `$nombre` con el valor "`Diego`".

Hay una diferencia en Perl entre los tres tipos de comillas posibles. Las comillas dobles (`"`) permiten la sustitución de variables en su interior. Por ejemplo, la línea siguiente:

```
print "El nombre es $nombre.";
```

imprimiría:

```
El nombre es Diego.
```

Sin embargo, las comillas simples (`'`) no permiten la sustitución de variables, con lo que lo siguiente:

```
print 'El nombre es $nombre.';
```

produciría:

```
El nombre es $nombre.
```

Por último, las comillas inversas (`` ` ``) significan "sustitución de la salida", al estilo de los distintos *shells*. Por ejemplo:

```
print "La fecha es ",`date`;
```

imprimiría:

```
La fecha es Thu Sep 10 17:08:18 CEST 1998
```

Pero las características que hicieron a Perl famoso fueron su facilidad para trabajar con *streams* y con cadenas de caracteres. Cualquier programa Perl posee tres *streams* estándar: `STDIN`, `STDOUT`, y `STDERR`, al igual que cualquier programa C. La lectura de un *stream* se realiza poniéndolo entre ángulos "`<>`":

```
$linea = <STDIN>;
print STDOUT $linea;
```

y como se ve, la escritura a través de `print`. Lo anterior lee una sola línea. si en vez de un escalar se hubiera utilizado una lista, todo el contenido del fichero se hubiera leido en la lista.

Para el manejo de cadenas, Perl ofrece un conjunto de operadores al estilo de `sed(1)`. Por defecto, éstos actúan sobre la variable `$_`, pero pueden ser aplicados sobre cualquier otra variable utilizando el operador "`=~`".

El operador "`s///`" sustituye en una cadena las ocurrencias de otra. Un ejemplo de su uso puede ser el siguiente:

```
$linea =~ s/hola/adiós/g;
```

Esto sustituye *todas* las ocurrencias de "`hola`" por "`adiós`". El modificiador "`g`" especifica que se deben sustituir todas. Existen otros modificadores como "`i`", que indica que se ignoren mayúsculas y minúsculas. Estas sustituciones se realizan sobre la variable `$linea`. Además, existen otros operadores como `m//`, `tr///`, etc.

Las construcciones de iteración son muy parecidas a las de C. Existen construcciones "`for`" y "`while`", además de otras muchas construcciones que permiten especificar bucles.

En cuanto a las funciones, se pueden especificar con "`sub`":

```
sub printArg {
    print join(",",@_);
}
```

La función se llama "`printArg`", y acepta un número no determinado de parámetros. Los parámetros se introducen en la lista especial "`@_`". La función, por lo tanto, imprime todos los argumentos con los que se la llame, separados por comas. Las llamadas a función se realizan anteponiendo el carácter "`&`" al nombre de la función:

```
&printArg("hola",2);
```

que imprime:

```
hola,2
```

En Perl existe la posibilidad de definir clases, llamar a métodos de objetos, etc., pero esto queda fuera del alcance de esta introducción. Existe, además, un conjunto muy grande de librerías para cualquier tarea imaginable, desde programación CGI hasta programación matemática.

B.2 Tcl/Tk

El lenguaje de programación Tcl fue concebido a finales de los 80 por el doctor John K. Ousterhout ([Ous90], [Ous98], [Fer98]). Fue ideado como un lenguaje sencillo que podía ser embebido en programas para facilitar la definición de tareas sencillas (el significado genérico de *scripting*).

La sintaxis de Tcl es muy sencilla. cualquier conjunto de símbolos forma una cadena, que es interpretada por el intérprete Tcl. Al igual que en Perl existen dos tipos de comillas, las dobles que permiten sustitución de variables y las sencillas que no, en Tcl existen las comillas dobles, que permiten sustitución al estilo de Perl, y las llaves ("`{}`"), cuyo contenido no es interpretado.

Las variables en Tcl se pueden asignar con el comando "`set`":

```
set var 3
```

establece el valor "`3`" a la variable "`var`". La sustitución de esta variable por su valor se consigue anteponiendo un "`$`" al nombre de la variable, por lo que, para imprimir su valor, se puede utilizar el comando "`puts`" de la siguiente manera:

```
puts "El valor es $var"          => El valor es 3
```

Sin embargo, utilizando llaves:

```
puts {El valor es $var}          => El valor es $var
```

Tcl es un lenguaje orientado a comandos. La sintaxis "`[comando argumentos]`" significa la ejecución del comando "`comando`" con los argumentos dados. Existen una serie de comandos incluidos en el intérprete, y otros nuevos pueden ser creados a través de librerías C o C++. Por ejemplo, el comando "`expr`" evalúa expresiones al estilo de `expr(1)`. Si queremos duplicar el valor de la variable "`var`", debemos escribir:

```
set var [expr $var * 2]
puts $var                        => 6
```

Nuevos comandos se pueden escribir en forma de procedimientos. Un procedimiento en Tcl consta de un nombre, unos argumentos y un cuerpo:

```
proc fact {x} {
    if {$x <= 1} then {
        return 1
    } else {
        return [expr $x * [fact [expr $x - 1]]]
    }
}

puts [fact 5]                    => 120
```

El procedimiento definido es "`fact`". Salvo por el uso de las llaves en el `if`, que en este caso significan la evaluación retrasada, la expresión "`[fact [expr $x - 1]]`" realiza una recursión normal. Nótese cómo, en la última línea, la invocación del procedimiento se realiza como una ejecución de comando, utilizando los corchetes.

Hay mucho más que decir de Tcl, sobre todo de su interfaz gráfico, Tk, pero hasta aquí llega esta introducción.

B.3 Python

Es increible la cantidad de desarrollos que se están realizando en torno a este lenguaje. En la sección 4.6 vimos incluso un ORB escrito completamente en Python. También existen servidores WEB, librerías matemáticas de gráficos, etc. Pero lo más sorprendente de este lenguaje es la extensa, completísima, fácil de utilizar y bien documentada librería estándar.

Python fue desarrollado a mediados de los 90 por Guido van Rossum del *Stichting Matematisch Centrum* en Amsterdam. Con la distribución en cada plataforma, viene una extensa documentación ([vR98d], [vR98c], [vR98b]). Las pretensiones en el diseño de Python van más allá de las de Tcl. Al igual que el primero, es posible insertarlo en aplicaciones en C o C++ como lenguaje de *script* y permite que nuevas ampliaciones se incorporen en el lenguaje. Sin embargo, a diferencia de Tcl, Python ofrece construcciones que permiten definir módulos, clases con herencia múltiple, tipado dinámico, metaclases, funciones de orden superior (funciones que aceptan como parámetro a otras funciones y funciones *lambda*), documentación integrada, etc.

Además, su sintaxis es a la vez sorprendente en principio y muy sencilla: se basa en la *indentación*. Un bloque queda subordinado a una construcción con tal de que comience en una columna más interna:

```
if x > 1:
    print x
    x = x - 1
```

Las dos últimas instrucciones quedan englobadas en el bloque que se ejecuta si `x` es mayor que 1. Su sintaxis tan homogénea, que los programadores en general se encuentran a gusto. Sobre todo los que vienen del mundo *Smalltalk*.

Python ofrece varios tipos de datos, literales, listas (que a su vez pueden contener otras listas), tablas de asociación y tuplas. El siguiente ejemplo define una tabla de asociación que asocia a las claves "`Diego`" y "`Juana`" una lista a cada una:

```
empleados = {'Diego': [1,2], 'Juana': [2,3]}
print empleados['Diego']                        => [1,2]
```

Las capacidades de este lenguaje son muy amplias. Por ejemplo, ofrece funciones `map` y `reduce` al estilo de los lenguajes declarativos, además de funciones *lambda*:

```
map( (lambda x: x+1), [1,2,3,4])                => [2,3,4,5]
```

La definición de clases y métodos es tan homogénea como todas las construcciones. Los atributos no son declarados en la definición de la clase, sino que son añadidos conforme se necesitan. El siguiente código define una calse que encapsula una referencia bibliográfica:

```
# -*- Python -*-
# ------------------------------------------------------------
class BibRef:
    def __init__(self,au,tit,year):
        self.author = au
        self.title = tit
        self.year = year

    def __repr__(self):
        # Asume un autor de la forma X. YYY
        return "[" + self.author[3:6] + self.year[2:] + "]"
# ------------------------------------------------------------
b = BibRef("D. Sevilla", "Algo", "1998")
print `b`                              => [Sev98]
```

La definición de la clase está encerrada entre las líneas de comentarios con "`-`". Se definen dos métodos especiales: "`__init__`", el constructor, y "`__repr__`", que retorna el objeto convertido en una cadena de caracteres. Este método es llamado cuando se requiere la representación en forma de cadena utilizando las comillas inversas, como en el ejemplo anterior. En el último método, se muestra el uso del operador *slice* ("`[:]`"), que "corta" trozos tanto de listas como de cadenas de caracteres, sirve para especificar rangos, etc.

El conjunto de paquetes estándar que provee Python es asombroso: programación CGI, programación matemática, interfaz con Tk (Tkinter), serialización de objetos, etc., etc. Referencias adicionales son [Mul96], [Sav97] y [vR98a].

Bibliografía

[AB94] L. J. Arthur and T. Burns. *UNIX Shell Programming.* John Wiley & Sons, Inc., third edition, 1994.

[APA98] Apache. *Apache Web Server Documentation*, 1998. http://www.apache.org.

[BJM94] P. Bohnhoff, R. Janssen, and R. Martín. *Fundamentos Cliente/Servidor.* IBM, 1994.

[BL94] T. Berners-Lee. *Universal Resource Identifiers in WWW, A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World Wide Web.* RFC 1630, 1994.

[Bro94] K. Brockschmidt. *Inside OLE 2.* Microsoft Press, 1994.

[C+96] R. Casselberry et al. *Running a Perfect Intranet.* Que Corporation, 1996.

[Chi98] M. Chilvers. *Fnorb, version 0.7.1*, June 1998. http://www.dstc.edu.au/Fnorb.

[CHY+97] P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C. Wang, and Y. Wang. Dcom and corba side by side, step by step and layer by layer. 1997. http://www.bell-labs.com/~emerald/dcom_corba/Paper.html.

[Cur97] D. Curtis. *Java, RMI and CORBA.* Object Management Group, 1997. http://www.omg.org/news/wpjava.htm.

[DG96] J. December and M. Ginsburg. *HTML 3.2 and CGI Professional Reference Edition UNLEASHED.* Macmillan Computer Publishing, 1996.

[FAS98] FAST, Inc. *Fast FTP Search Version 2.4*, 1998. http://ftpsearch.ntnu.no.

[FB96] N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME).* RFC 2045, 2046, 2047 y 2048, 1996.

[Fer98] A. Ferrieux. *Concepts of Architectural Design for Tcl Applications*, July 1998. ftp://ftp.neosoft.com/pub/tcl/sorted/packages-7.6/info/doc/tclarch.txt.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2068, 1997.

[Fri96] Æ. Frish. *Essential System Administration.* O'Reilly & Associates, Inc., second edition, 1996.

[Gar98] L. M. Garshol. *What's wrong with Perl*, 1998. http://www.stud.ifi.uio.no/~lars-ga/download/artikler/perl.html.

[Hag96] J. Hagey. *PC Magazine Programming Perl 5.0 CGI Web Pages for Microsoft Windows NT*. Macmillan Computer Publishing, 1996.

[HB96] K. Husain and R. F. Breedlove. *Perl 5 UNLEASHED*. Sams Publishing, 1996.

[HER98] Hermetica. ***DBI*** *drivers and module pages*, 1998. http://www.hermetica.com/tecnologia/perl/DBI.

[Jon96] C. Jones. *Programming Languages Table, Release 8.2*, March 1996. http://www.spr.com/library/0langtbl.htm.

[Jon97] C. Jones. *What Are Function Points?*, 1997. http://www.spr.com/library/0funcmet.htm.

[JWS97] Sun Microsystems. *Java Web Server 1.1 Documentation*, 1997. http://jserv.javasoft.com/products/java-server/documentation/index.html.

[KP87] B. W. Kernighan and R. Pike. *El entorno de programación UNIX*. Prentice Hall Hispanoamericana, S. A., 1987.

[Lew95] T. G. Lewis. Where is client/server software headed? *IEEE Computer*, April 1995.

[LH97] Z. P. Lazar and P. Holfelder. Cgi vs. server-side javascript for database applications. *Netscape View Source Magazine*, July 1997.

[LS97] C. Laird and K. Soraiz. Choosing a scripting languaje. *SunWorld*, October 1997.

[Mic95a] Microsoft Corp. *The Component Object Model (COM) Specification*, 1995. Draft Version 0.9.

[Mic95b] Microsoft Corporation. *Microsoft Open Database Connectivity™ Software Development Kit, Version 2.10*, 1995.

[Mic96] Microsoft Corp. *DCOM Technical Overview*, 1996. White Paper, Microsoft Developer Network, April 1997.

[Moh98] P. Mohseni. Exploit distributed java computing with rmi. *NC World*, January 1998. http://www.ncworldmag.com.

[Mue96] J. Muelver. *Creación de Páginas* Web *con* Perl. Anaya Multimedia, S.A., 1996.

[Mul96] S. Mullender. *Comparison of Tcl and Python*, 1996. http://www.cwi.nl/~sjoerd/PythonVsTcl.html.

[MZ95] T. J. Mowbray and R. Zahavi. *The Essential CORBA*. John Wiley & Sons, Inc., 1995.

[Net98] Netscape Corp. *JavaScript Working Document*, 1998.

[Nic96] A. Nicolaou. *A survey of distributed languages*, 1996. http//www.cgl.uwaterloo.ca/~anicolao/termpaper.html.

[OH97] R. Orfali and D. Harkey. *Client/Server Programming with JAVA™ and CORBA*. John Wiley & Sons, Inc., 1997.

[OHE96] R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.

[OMG97] Object Management Group. *Common Object Request Broker Architecture and Specification, revision 2.1*, August 1997.

[OMG98] Object Management Group. *CORBA Main Site*, 1998. http://www.omg.org.

[Ous90] J. K. Ousterhout. Tcl: An embeddable command language. In *Winter USENIX Conference Proceedings*, 1990.

[Ous98] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, March 1998. También en: http://www.scriptics.com/people/john.ousterhout/scripting.html.

[PHP98] PHP Development Team. *PHP3 Documentation*, 1998. http://www.php.net.

[Pos82] J. B. Postel. *Simple Mail Transfer Protocol*. RFC 822, 1982.

[PWGB98] D. Pedrick, J. Weedon, J. Goldberg, and E. Bleinfield. *Programming with VisiBroker™*. John Wiley & Sons, Inc., 1998.

[Ros97] J. Rosenberger. *Teach Yourself* CORBA *in 14 days*. Macmillan Computer Publishing, 1997.

[ROT98] Roth Intl. `Win32::ODBC` *package and documentation*, 1998. http://www.roth.net.

[Sav97] V. Savikko. Design patterns in python. In *Proceedings of the 6th International Python Conference*, October 1997.

[SE94] B. Stroustrup and M. A. Ellis. *C++, Manual de Referencia con Anotaciones*. Addison-Wesley Iberoamericana, S.A., 1994.

[Sho97] M. Shoffner. Increase the functionality in your distributed client/server apps. *Java World*, October 1997. http://www.javaworld.com.

[Sho98] M. Shoffner. Networking our whiteboard with servlets. *Java World*, January 1998.

[Ste90] W. R. Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[Ste98] L. D. Stein. *The World Wide Web Security FAQ*. http://www-genome.wi.mit.edu/WWW/faqs/www-security-faq.html, 1998.

[SUN98] SUN Microsystems. *The JDK 1.1.5 Documentation*, 1998. http://java.sun.com/products/jdk/1.1/docs.

[SV95a] D. C. Schmidt and S. Vinoski. Object interconections: Comparing alternative client-side programming techniques. *SIGS C++ Report*, May 1995.

[SV95b] D. C. Schmidt and S. Vinoski. Object interconections: Modeling distributed objects applications. *SIGS C++ Report*, Feb 1995.

[VD98] A. Voguel and K. Duddy. *JAVA™ Programming with CORBA*. John Wiley & Sons, Inc., 1998.

[VIS97a] Visigenic Software, Inc. *Visigenic's VisiBroker for Java Installation and Administration Guide, version 3.1*, November 1997.

[VIS97b] Visigenic Software, Inc. *Visigenic's VisiBroker for Java Programmer's Guide, version 3.0*, September 1997.

[vR98a] G. van Rossum. Java and python: a perfect couple. *developer.com Journal*, August 1998.

[vR98b] G. van Rossum. *Python Library Reference, release 1.5.1*, April 1998. http://www.python.org.

[vR98c] G. van Rossum. *Python Reference Manual, release 1.5*, March 1998. http://www.python.org.

[vR98d] G. van Rossum. *Python Tutorial, release 1.5.1*, April 1998. http://www.python.org.

[W3C97] W3 Consortium. *Issues in the Development of Distributed Hypermedia Applications*, 1997. http://www.w3.org/OOP/HyperMediaDev.

[Won97] C. Wong. *Web Client Programming with Perl.* O'Reilly & Associates, Inc., 1997.

[WS92] L. Wall and R. L. Schwartz. *Programming perl.* O'Reilly & Associates, Inc., 1992.

[Wut96] M. Wutka. *Hacking Java™: The Professional's Resource Kit.* Macmillan Computer Publishing, 1996.

Índice de Materias