4. SERVIDOR DE APLICACIONES

Hasta el momento, la mayor parte de las tecnologías que se han ido introduciendo no han hecho otra cosa que engordar y engordar el servidor Web, cargándolo con todo el trabajo adicional a base de parches o añadidos de diversa naturaleza —justo en contra de la idea original, donde un protocolo muy básico y una sencilla aplicación servidora gestionaban la presentación de contenidos a, potencialmente, cualquier equipo que se pudiera conectar a la red y dispusiera de un sencillo navegador Web.

A medida que las primeras y sencillas páginas Web se han transformado en sofisticadas «aplicaciones Web» que proporcionan soporte a organizaciones de cualquier tamaño, parece una mejor opción devolver este escenario a su estado inicial y buscar soluciones más ajustadas a los nuevos requerimientos. Según esto, para gestionar todas las posibles tareas de índole dinámica que un servidor Web deba realizar, se propone la colocación, *detrás de él,* un nuevo sistema especialmente concebido para resolverlas con solvencia y liberar al servidor Web de toda carga extra. Es lo que se conoce como «servidor de aplicaciones».

Este enfoque no implica que no se vayan a poder seguir empleando todas las tecnologías anteriormente estudiadas, sino que la tendencia será ir migrando las aplicaciones hacia la utilización del servidor de aplicaciones a medida que vayan creciendo los requerimientos de escalabilidad, eficiencia y adaptabilidad, por mencionar algunos de ellos, de las nuevas aplicaciones Web.

La filosofía general de funcionamiento del nuevo tándem servidor Web más servidor de aplicaciones es la siguiente: las peticiones realizadas al servidor Web que deban generar contenido estático serán despachadas por el propio servidor Web, como de costumbre, pero las peticiones que deban generar contenido dinámico serán delegadas en el servidor de aplicaciones; éste interactuará con los recursos que se precisen, ejecutará la

lógica de negocio asociada y le pasará la respuesta en formato HTML o XML al servidor Web que, a su vez, la remitirá al cliente que invocó la petición (ver figura 4.1).

Sin embargo, todavía podemos llegar un poco más lejos, de hecho, hasta la situación actual, donde nos encontramos con el que ha resultado ser el escenario más adecuado para desplegar complejas aplicaciones sobre Internet: estamos hablando de los «sistemas distribuidos».

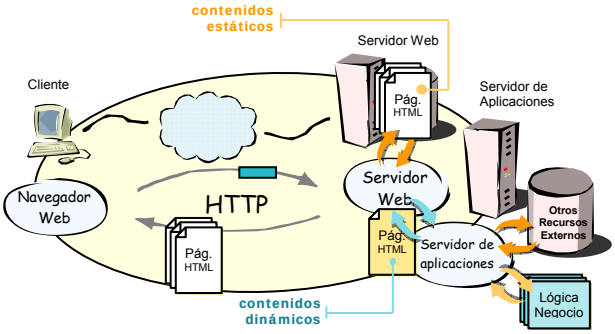

Figura 4.1. Relación entre un *servidor Web* y un *servidor de aplicaciones*.

En este tipo de escenarios, las aplicaciones se desarrollan como un conjunto de componentes software que encapsulan la lógica de negocio y que se ejecutan distribuidos sobre un conjunto de servidores de aplicaciones interconectados a través de una red de comunicaciones basada, generalmente, en TCP/IP (una intranet o, incluso, la propia Internet).

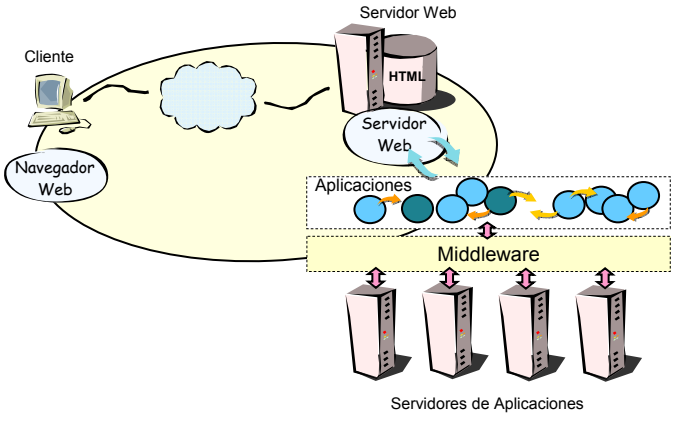

Figura 4.2. Capa de *middleware* en el contexto de un sistema distribuido

En un entorno de estas características seguimos interesados en perfeccionar el modelo de componentes pero, sobre todo, estamos más que interesados en que la complejidad de las infraestructuras que tienen que servir de soporte quede oculta para las aplicaciones y para los desarrolladores, y que su gestión no se convierta en un verdadero calvario para los administradores de sistemas.

De igual forma que en un equipo aislado es el sistema operativo el responsable de ocultar a las aplicaciones los detalles concretos de los dispositivos y componentes físicos, el conjunto de tecnologías y servicios que proporcionan la transparencia deseada (ver figura 4.2), ya no sólo sobre un único equipo, sino sobre todo un conjunto de equipos, así como de la propia red de comunicaciones e, incluso, de los sistemas operativos de cada una de ellos, lo denominamos «middleware».

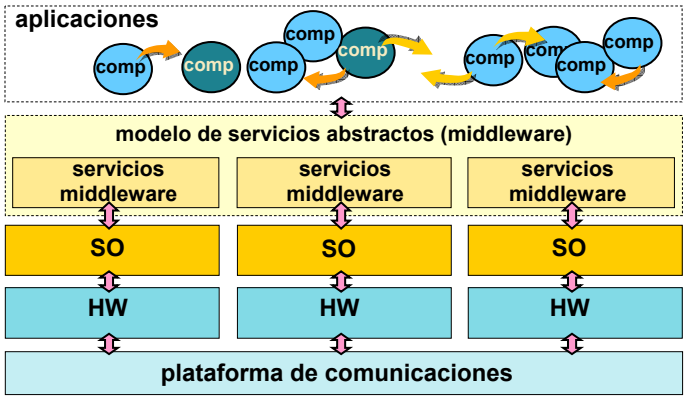

Figura 4.3. Modelo general de capas del sistema con middleware.

4.1. MIDDLEWARE

El middleware se define como el nivel lógico del sistema que proporciona una abstracción sobre la infraestructura que le da soporte a las aplicaciones, dotando de transparencia de ubicación e independencia de los detalles del hardware de los computadores, del sistema operativo, de la red de interconexión y de los protocolos de comunicaciones (figura 4.3) e, incluso, del lenguaje de programación utilizado para su desarrollo. Desde el punto de vista del programador de aplicaciones, el middleware establece un modelo de programación sobre bloques básicos arquitectónicos, utilizando protocolos basados en mensajes para proporcionar abstracciones de nivel superior.

En cualquier caso, el middleware se puede estudiar desde dos puntos de vista bien diferenciados: desde el punto de vista de la infraestructura y servicios que lo conforman o desde el punto de vista del desarrollador de aplicaciones. Veamos a continuación cada uno de ellos.

4.1.1. Infraestructura de servicios

Desde el punto de vista de la propia infraestructura que conforma el *middleware*, éste viene definido por los elementos que componen el núcleo de servicios, ubicado entre el sistema operativo local y las aplicaciones, y en la que resaltan los siguientes elementos (figura 4.4): el modelo de representación de datos, el modelo de comunicaciones —en el que se incluye el protocolo de comunicaciones y el modelo de invocación de métodos remotos— y los servicios fundamentales que proporciona la plataforma.

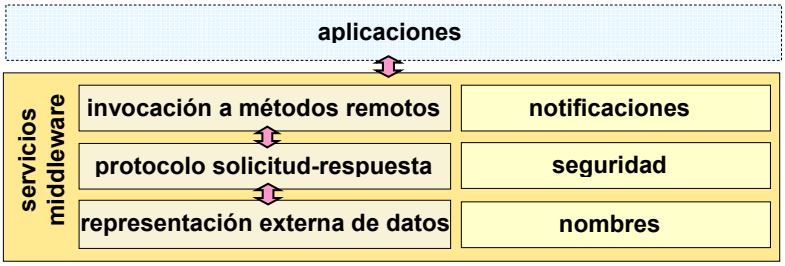

Figura 4.4. Alguno de los principales componentes que forman la capa de servicios middleware que proporciona soporte y transparencia a las aplicaciones.

4.1.1.1. Modelo de representación de datos

El modelo de comunicación proporciona los mecanismos para que dos aplicaciones potencialmente ubicadas en equipos diferentes puedan interactuar entre sí de forma transparente. Al mismo tiempo, estos mecanismos determinan cómo se deberá establecer la comunicación y cómo tendrán que prepararse cliente y servidor para poder realizarla.

Independientemente de la forma de comunicación utilizada, las estructuras de datos y los atributos de los objetos deben ser convertidos en una secuencia de *bytes* antes de su transmisión y reconstruidos posteriormente en el destino. Para hacer posible que dos computadores puedan intercambiar información, deben acordar previamente un esquema común o incluir la especificación del formato de emisión en el propio mensaje.

Al estándar acordado para la representación de los valores y estructuras de datos a transmitir en los mensajes se denomina *representación externa de datos* —por ejemplo, CDR de CORBA.

4.1.1.2. Modelo de comunicación

El protocolo petición-respuesta define el nivel adecuado para establecer una comunicación típica según la arquitectura cliente/servidor en sistemas distribuidos basados en UDP o TCP. El propio mensaje de respuesta se considera como un reconocimiento del mensaje de petición, evitando de este modo las sobrecargas de los mensajes de reconocimiento adicionales.

Sobre esta capa de protocolo se construyen los modelos de comunicación de llamada a procedimiento remoto (RPC) y el modelo de invocación a un método remoto (RMI). RPC permite a programas cliente invocar procedimientos pertenecientes a programas servidor que generalmente ejecutan en computadores distintos. Este modelo evoluciona, posteriormente, permitiendo que objetos de diferentes procesos se comuniquen con los métodos de otros objetos.

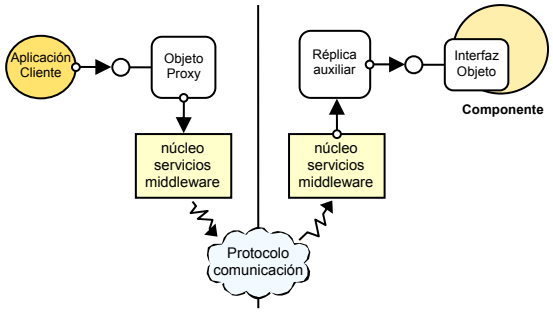

Figura 4.5. Arquitectura de comunicaciones middleware.

En la figura 4.5 se puede observar una arquitectura de comunicaciones más o menos genérica de un middleware. Cuando se crea un componente que debe ser accedido de forma remota, debe crearse también un objeto *proxy* de la interfaz de dicho componente. De esta forma, el middleware puede presentar a la aplicación cliente una interfaz uniforme, con independencia de dónde se encuentre realmente el componente al que desea acceder, encargándose de resolver todos los problemas de redirección de las solicitudes y respuestas, de localización de los componentes o de que los componentes se encuentran cargados en memoria y listos para responder.

4.1.1.3. Servicios comunes

Para las aplicaciones que se ejecutan apoyándose en el middleware, éste representa fundamentalmente un conjunto de servicios a los que pueden acceder o a partir de los cuales pueden acceder a otros servicios y recursos. El mecanismo de comunicación estudiado en el punto anterior es uno de estos servicios, quizá uno de los más importantes, pero no el único. De hecho, cada fabricante propondrá un conjunto de servicios que estime convenientes y que no tienen por qué coincidir de un middleware a otro. Por suerte, existe una serie de servicios que se pueden encontrar de una forma u otra en las diferentes propuestas de middleware para componentes software distribuidos. A continuación se realiza una descripción breve de los más comunes: servicios de nombres, de eventos y notificaciones y de seguridad.

Servicio de nombres

Su objetivo es almacenar los atributos de los objetos en un sistema distribuido —nombre y dirección entre otros—, devolviendo esos atributos cuando se realiza una búsqueda sobre cierto objeto. Los principales requisitos que debe poseer un servicio de nombres son: habilidad para manejar un número arbitrario de nombres, persistencia, alta disponibilidad y tolerancia a fallos. Ejemplos de este tipo de servicio son: X.500, Jini, DNS, LDAP, etc.

Servicio de eventos y notificaciones

Este servicio extiende el modelo local de eventos al permitir que varios objetos en diferentes ubicaciones puedan ser notificados de los eventos que tienen lugar en un objeto. Emplean el paradigma *del tablón de anuncios,* en el que un objeto que genera eventos publica el tipo de eventos que ofrece para su observación. Los objetos que desean recibir notificaciones de otro objeto se suscriben a los tipos de eventos que les interesan.

Servicio de seguridad

Los mecanismos de seguridad se basan esencialmente en la criptografía de clave pública y de clave secreta. Los recursos se protegen mediante mecanismos de control de acceso. Se pueden mantener los derechos de acceso en listas de control (ACL) asociadas a conjuntos de objetos. Ejemplos de este servicio son: el protocolo de autenticación Needham-Schroeder, Kerberos, SSL y Millicent.

4.1.2. Modelo de programación y de componentes

Desde el punto de vista del diseñador de aplicaciones, el middleware representa un modelo de programación al que le incumbe definir nítidamente la estructura que deben tener los componentes software a diseñar, así como los mecanismos que deben emplearse para acceder a los recursos y a los servicios que proporciona la plataforma.

Los componentes software se pueden considerar piezas de software diseñadas para ser reutilizadas en diferentes planos: reutilización de código, de funcionalidad, de tipos. Los componentes software distribuidos, además, hacen especial hincapié en su ejecución transparente sobre entornos distribuidos. Están creados para encapsular una determinada lógica de negocio, un determinado servicio (como puede ser acceso a una determinada arquitectura) o, incluso, ser empleados como almacén de datos. Estos componentes se utilizan en la construcción de aplicaciones como si fueran piezas de un mecano.

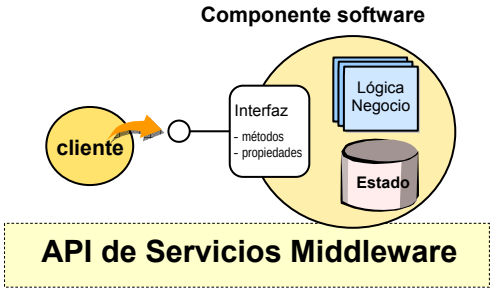

Figura 4.6. Estructura básica de un componente software y su relación con la plataforma.

Todas estas características resultan fundamentales para diseñar, desarrollar y mantener las actuales aplicaciones Web, sobre todo en el ámbito empresarial. Gracias a ellas se pueden aplicar técnicas de ingeniería del software sofisticadas y se facilita enormemente la adaptación de la aplicación a la situación cambiante del entorno. Una aplicación distribuida basada en el modelo de middleware permite una asignación de recursos muy ajustada a medida que éstos vayan necesitándose y, siempre que los cambios puedan localizarse en unos componenetes software determinados, su modificación tiene un impacto relativamente escaso sobre el global de la aplicación.

Un componente software debe poseer una estructura bien definida (ver fig. 4.6): una *interfaz* en la que se definen los métodos y propiedades que soporta el componente y a través de la cual los clientes pueden acceder;

una *lógica de negocio* o lógica de aplicación que recoge la funcionalidad o el comportamiento del componente, tanto la que muestra al exterior a través de su interfaz, como la que presenta internamente; y un *estado* que le permite almacenar información que permite que dos instancias de un mismo componente puedan adquirir su propia entidad.

Por otra parte, el modelo de programación también determina cómo accederán estos componentes software a los servicios que ofrece el middleware y que, en este caso, consiste en la provisión de una *interfaz de programación (API) de servicios.*

4.1.2.1. Estructura de un componente

Un sistema orientado a objetos consta de un conjunto de componentes que interaccionan entre sí, cada uno de los cuales consiste en un conjunto de propiedades y un conjunto de métodos. Un objeto se comunica con otro objeto invocando sus métodos, generalmente pasándole argumentos y recibiendo resultados. Así, cada componente se puede estructurar en tres elementos: una interfaz, una lógica de aplicación o de negocio y un estado:

Interfaz

La interfaz de un componente proporciona una definición de las signaturas de sus métodos —los tipos de sus argumentos, valores devueltos, excepciones, etc.— sin especificar su implementación. Establece, asimismo, qué métodos y propiedades están accesibles para el resto de componentes y objetos. Por supuesto, nada impide que un mismo objeto implemente distintas interfaces a la vez.

Cada componente se implementa de forma que oculta todo su estado y funcionalidad, excepto aquél que se hace visible a través de su interfaz en términos de *propiedades* y *métodos*, respectivamente. Esto permite modificaciones en la implementación de la lógica del componente o de su estructura interna —siempre que su interfaz se mantenga inalterada— con un impacto mínimo sobre el resto de la aplicación de la que forma parte.

Para que distintos componentes —implementados por distintos equipos de desarrollo y, posiblemente, con diferentes lenguajes de programación— puedan interactuar entre sí es necesario que sus interfaces estén definidas de forma homogénea. Para facilitar esto, el modelo de programación introduce los lenguajes de definición de interfaces (IDL) que proporcionan una notación específica en la que, por ejemplo, los parámetros de un método se describen como de entrada o salida y utilizando su propia especificación de tipos.

Estado
El estado de un componente consta de los valores de sus variables de instancia. En el paradigma de la programación orientada a objetos, el estado de un programa se encuentra fraccionado en partes separadas, cada una de las cuales está asociada con un objeto. El estado de un objeto está accesible sólo para los métodos del objeto, es decir, que no es posible que métodos no autorizados actúen sobre su estado.

Algunas implementaciones de middleware, como CORBA, permiten empaquetar y almacenar los objetos junto con su estado en un momento dado —en los llamados almacenes de objetos persistentes—, de forma que métodos de otros objetos puedan activarlos por invocación a través de su interfaz. En general, se almacenan en cualquier momento en el que se encuentren en un estado consistente, con lo que dotan al sistema de tolerancia a fallos.

Lógica de negocio o comportamiento
El comportamiento define la funcionalidad del componente a través de una serie de métodos que recogen la lógica de aplicación o de negocio que encapsula y a la cual se puede acceder, en caso de estar disponible para otros componentes, a través de las interfaces definidas.

4.1.2.2. Acceso a los servicios

Una plataforma middleware debe proporcionar un mecanismo que permita a los componentes software acceder a los servicios que proporciona. El método más habitual, sobre todo con el objetivo de dotar a los desarrolladores de una independencia adecuada con respecto a los lenguajes y herramientas de programación que utilicen, es a través de interfaces de programación de aplicaciones (API).

4.2. PLATAFORMAS ACTUALES

Aunque desde el punto de vista más amplio podríamos definir un *servidor de aplicaciones* como «uno o más servidores de red, dedicados a ejecutar ciertas aplicaciones software, de forma que sus componentes pueden comunicarse entre sí de forma transparente», el término también hace referencia al propio software instalado en estos servidores cuyo objetivo es servir de soporte para la ejecución de las aplicaciones mencionadas. Es por esta función de soporte por la que este software de apoyo se denomina también *plataforma middleware*.

En la práctica, una de las plataformas que más ha contribuido a esta visión es J2EE de *Sun Microsystems*, razón por la cual en muchos entornos se considera un sinónimo de *servidor de aplicaciones*. Sin embargo, puesto que J2EE en realidad es un conjunto de especificaciones, en la actualidad podemos encontrar múltiples alternativas de implementación de otros fabricantes y, lo que es más interesante, otras especificaciones válidas.

Los servidores de aplicación típicamente incluyen el middleware que les permite intercomunicarse con diferentes servicios de forma segura y proporcionan a los desarrolladores un API que aísla a las aplicaciones del sistema operativo del servidor y brindan soporte a una gran variedad de estándares, tales como HTML, XML, IIOP, JDBC, SSL, etc., que facilitan la interoperatividad de elementos heterogéneos y la conexión a una gran variedad de fuentes de datos, sistemas y dispositivos.

A continuación se abordan algunas de las plataformas que están más extendidas en la actualidad y que proporcionan estos servicios middleware.

4.2.1. Plataforma J2EE

J2EE son las siglas de *Java 2 Enterprise Edition*, es decir, la edición empresarial del paquete Java creada y distribuida por *Sun Microsystems*. Comprende un conjunto de especificaciones y funcionalidades orientadas al desarrollo de aplicaciones empresariales. Debido a que J2EE no deja de ser la definición de un estándar, existen otros productos desarrollados a partir de la misma, aunque no exclusivamente.

Algunas de las funcionalidades más importantes que aporta esta plataforma son las siguientes:

- Proporciona acceso a bases de datos (JDBC).
- Disponibilidad de implementaciones de la plataforma por diferentes fabricantes: BEA, IBM, Oracle, Sun, y Apache Tomcat entre otros.
- Permite la utilización de directorios distribuidos (JNDI).
- Proporciona acceso a la invocación de métodos remotos (RMI, CORBA).
- Dispone de funciones de correo electrónico (JavaMail).
- Aplicaciones Web (JSP y Servlet).
- Puede emplear componentes software como: Beans, objetos CORBA, etc.

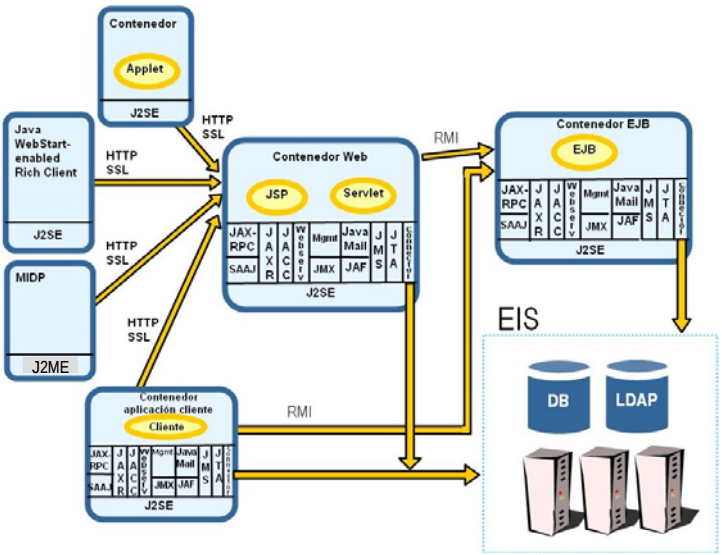

Figura 4.7. Arquitectura J2EE.

La Plataforma J2EE desarrolla el concepto de contenedores a partir de la tecnología *Java 2 Estándar Edition* (J2SE) propuesta también por Sun Microsystems, integra los elementos existentes (JSP, applet, servlet, Javabeans), lo extiende al de *Enterprise Java Bean* (EBJ) como modelo de componentes software distribuidos y basa la comunicación en el estándar RMI (*Remote Methode Invocation*) tratado en temas anteriores.

En la figura 4.7 se presenta un diagrama con los componentes que forman parte de esta plataforma. Se puede observar cómo los diferentes objetos del sistema (JSP, servlet, Applet o EJB) se ejecutan en *contendores* concretos: de aplicación cliente, Web, EJB y navegador Web compatible. Todos los contenedores están basados en J2SE como núcleo básico. Se puede encontrar una excepción en el caso de dispositivos móviles o sistemas embebidos de muy poca capacidad, para los cuales se ha tenido que proponer y desarrollar un núcleo de J2SE mucho más compacto y ligero que se denomina *Java 2 Mobile Edition* (J2ME). Sobre este núcleo, se agregan aquéllos servicios que puedan resultar útiles a las aplicaciones que se ejecuten en cada contenedor (como JAX-RPC, SAAJ, JTA, JAF o JMX; detallados más adelante dentro de este mismo punto) junto con una serie de conectores que facilitan el acceso de las aplicaciones a sistemas de información empresarial (bases de datos, sistemas heredados, etc.).

4.2.1.1. Servicios midlleware en J2EE

J2EE propone el acceso a los servicios que define a través de una interfaz de programación. Las principales APIs de la plataforma J2EE de SUN son las siguientes:

- **JCA** Arquitectura que para interactuar con una variedad de EIS, incluye ERP, CRM y otra serie de sistemas heredados.
- **JDBC** Acceso a base de datos relacionales.
- **JTA** Manejo y la coordinación de transacciones a través de EIS heterogéneos.
- **JNDI** Acceso a información en servicios de directorio y servicios de nombres.
- **JMS** Envío y recepción de mensajes.
- **JMail** Envío y recepción de correo.
- **JIDL** Mecanismo para interactuar con servicios CORBA.

Por supuesto, se dispone de muchos otros APIs orientados a aspectos como: tratamiento de XML, integración con sistemas heredados utilizando Servicios Web, etc.

4.2.1.2. Enterprise JavaBeans

Los EJBs proporcionan un modelo de componentes software distribuido normalizado para el lado del servidor. El objetivo de los Enterprise beans es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial (concurrencia, transacciones, persistencia, seguridad, ...) para centrarse en el desarrollo de la lógica de negocio en sí. El hecho de estar basado en componentes nos permite que éstos sean flexibles y, sobre todo, reutilizables.

No hay que confundir a los *Enterprise JavaBeans* con los *JavaBeans*. Los JavaBeans también son un modelo de componentes creado por *Sun Microsystems* para la construcción de aplicaciones, pero no pueden utilizarse en entornos de objetos distribuidos al no soportar nativamente la *invocación a métodos remotos* (RMI).

El API Enterprise JavaBeans (EJB) permite escribir componentes software en Java, lo que les confiere la funcionalidad de portabilidad e independencia de la plataforma, a diferencia de los componentes ActiveX o COM+, propios de los sistemas Windows, o los VCL, asociados a los entornos de desarrollo Delphi, por citar los más conocidos.

Existen tres tipos de EJBs:

- **EJBs de Entidad** (*Entity EJBs*): su objetivo es encapsular los objetos de lado de servidor que almacenan los datos. Los EJBs de

entidad presentan la característica fundamental de la persistencia, ya sea gestionada por el propio contenedor (CMP) o por el bean (BMP).

- **EJBs de Sesión** (*Session EJBs*): gestionan el flujo de la información en el servidor. Generalmente sirven a los clientes como una fachada de los servicios proporcionados por otros componentes disponibles en el servidor. Puede haber dos tipos: «con estado» (*stateful*) o «sin estado» (*stateless*).
- **EJBs dirigidos por mensajes** (*Message-driven EJBs*): los únicos beans con funcionamiento asíncrono. Usando el *Java Messaging System* (JMS), se suscriben a un «tópico» (*topic*) o a una «cola» (*queue*) y se activan al recibir un mensaje dirigido a dicho tópico o cola. No requieren de su instanciación por parte del cliente.

4.2.1.3. Funcionamiento de un Enterprise JavaBean

Los EJBs se disponen en un *contenedor EJB* dentro del *servidor de aplicaciones*. La especificación describe cómo el EJB interactúa con su contenedor y cómo el código cliente interactúa con la combinación del EJB y el contenedor.

Cada EJB debe facilitar una clase de implementación Java y dos interfaces Java. El contenedor EJB creará instancias de la clase de implementación Java para facilitar la implementación EJB. Las interfaces Java son utilizadas por el código cliente del EJB. Las dos interfaces, conocidas como «interfaz home» e «interfaz remota», especifican las firmas de los métodos remotos del EJB.

El servidor invocará a un método correspondiente a una instancia de la clase de implementación Java para manejar la invocación del método remoto. Las llamadas a métodos en la interfaz remota se remiten al método de implementación correspondiente del mismo nombre y argumentos

Dado que se trata simplemente de interfaces Java y no de clases concretas, el contenedor EJB genera clases para esas interfaces que actuarán como intermediarias —o como un «proxy»— entre los clientes y los componentes. El cliente invoca un método en los proxies generados que, a su vez, sitúa los argumentos método en un mensaje y envía dicho mensaje al servidor EJB. Los proxies usan RMI-IIOP para comunicarse con el servidor EJB.

RMI es el mecanismo que permite realizar invocaciones a métodos de objetos remotos situados en distintas (o en la misma) máquinas virtuales de Java, compartiendo así recursos y carga de procesamiento a través de varios sistemas.

La arquitectura RMI puede verse como un modelo de cuatro capas (ver figura 4.8):

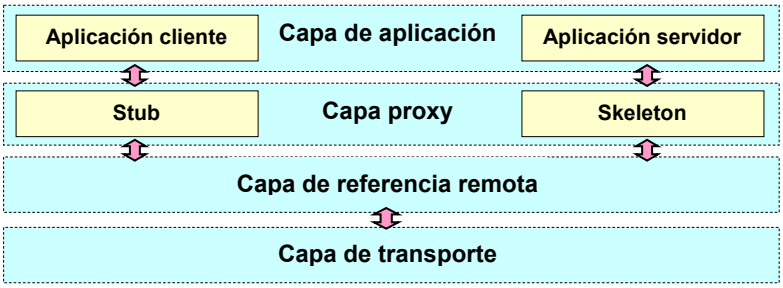

Figura 4.8. Arquitectura RMI en la que J2EE basa la comunicación de sus componentes.

La primera capa es la de aplicación y se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda `java.rmi.Remote`. Dicha interfaz se usa básicamente para *marcar* un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase `UnicastRemoteObject` (paquete `java.rmi.server`), o puede hacerse de forma explícita con una llamada al método `exportObject()` del mismo paquete.

La segunda capa es la capa proxy, o capa *stub-skeleton*. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

La tercera capa es la de referencia remota, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo *stream* (*stream-oriented connection*) desde la capa de transporte.

La cuarta y última capa es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es

JRMP (*Java Remote Method Protocol*), que solamente es válido para aplicaciones Java.

Toda aplicación RMI normalmente se descompone en 2 partes:

- Un servidor, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- Un cliente, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

4.2.1.4. Servidores de aplicación J2EE

Sun Microsystems, con su servidor de aplicaciones *Sun Java System Application Server,* no es la única que ha desarrollado su especificación J2EE. Bajo «certificados oficiales de compatibilidad», fabricantes diversos han planteado sus propias implementaciones y pueden garantizar que se ajustan completamente al estándar J2EE. *WebSphere* (*IBM*), *Oracle Application Server* (*Oracle Corporation*) y *WebLogic* (*BEA*) están entre los servidores de aplicación J2EE propietarios más conocidos. *EAServer* (*Sybase Inc.*) es también conocido por ofrecer soporte a otros lenguajes diferentes a Java, como PowerBuilder. El servidor de aplicaciones *JOnAS*, desarrollado por el consorcio *ObjectWeb*, fue el primer servidor de aplicaciones libre en lograr certificación oficial de compatibilidad con J2EE. *Tomcat* (*Apache Software Foundation*) y *JBoss* son otros servidores de aplicación libres muy populares en la actualidad.

La portabilidad de Java también ha permitido que los servidores de aplicación J2EE se encuentren disponibles sobre una gran variedad de plataformas, como *Microsoft Windows*, *Unix* y *GNU/Linux*.

4.2.2. Plataforma .NET

.*NET* es un proyecto de *Microsoft* para crear una nueva plataforma de desarrollo de software con énfasis en transparencia de redes, con independencia de plataforma y que permita un rápido desarrollo de aplicaciones.

.*NET* podría considerarse una respuesta de *Microsoft* al creciente mercado de los negocios en entornos Web, como competencia a la plataforma Java de *Sun Microsystems*.

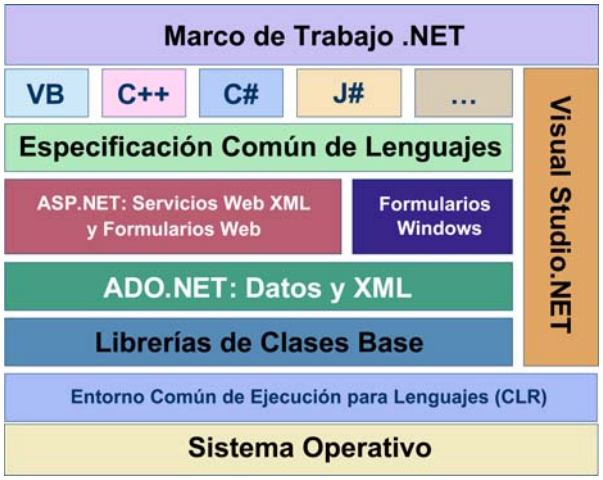

Figura 4.9. Componentes de .NET Framework.

4.2.2.1. .NET Framework

El «framework» o marco de trabajo .NET constituye la base de la plataforma .NET y denota la infraestructura sobre la cual se reúnen un conjunto de lenguajes, herramientas y servicios que simplifican el desarrollo de aplicaciones en entornos de ejecución distribuidos. Este marco también reúne una serie de normas impulsadas por varias compañías además de Microsoft (como Hewlett-Packar, Intel, IBM, Fujitsu Software, Plum Hall, la Universidad de Monash e ISE).

Los principales componentes del marco de trabajo son:

- El conjunto de lenguajes de programación.
- La Biblioteca de Clases Base o BCL.
- El Entorno Común de Ejecución para Lenguajes (CLR).

Debido a la publicación de la *norma para la infraestructura común de lenguajes* (*Common Language Infraestructure*, CLI), se ha facilitado enormemente el desarrollo de lenguajes de programación compatibles, por lo que el marco de trabajo .NET soporta ya más de 20 lenguajes de programación y es posible desarrollar cualquiera de los tipos de aplicaciones soportados en la plataforma con cualquiera de ellos, lo que elimina las diferencias que existían entre lo que era posible hacer con uno u otro lenguaje.

Algunos de los lenguajes desarrollados para el marco de trabajo .NET son: *C#*, *Visual Basic*, *Turbo Delphi for .NET*, *C++*, *J#*, *Perl*, *Python*, *Fortran* y *Cobol.NET*.

4.2.2.2. Common Language Runtime

El entorno común para la ejecución (*Common Language Runtime*, CLR) es el verdadero núcleo del marco de trabajo .NET. El CLR es el entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes, ampliando el conjunto de servicios del sistema operativo.

La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un código intermedio (*Microsoft Intermediate Lenguage*, MSIL), similar al *BYTECODE* de Java. Para generar dicho código el compilador se basa en el *Common Language Specification* (CLS) que determina las reglas necesarias para crear ese código MSIL compatible con el CLR.

Para ejecutarse se necesita un segundo paso. Un compilador JIT (*Just-In-Time*) genera el código máquina real que se ejecuta en la plataforma del cliente; de esta forma se consigue con .NET independencia de la plataforma hardware.

Figura 4.10. Diagrama de la estructura interna del CLR.

La compilación JIT la realiza el CLR a medida que el programa invoca métodos. El código ejecutable generado se almacena en la memoria caché del ordenador, siendo recompilado de nuevo sólo en el caso de producirse algún cambio en el código fuente.

4.2.2.3. Objetos COM+

Bajo las siglas COM+ (*Common Object Model Plus*) se define el modelo de componentes software basados en servicios, propuesto por Microsoft que combina, extiende y unifica los servicios propuestos por los anteriores modelos COM, DCOM y MTS:

- COM (*Component Object Model*). Modelo de objeto componente. Arquitectura software de componentes de Microsoft diseñada para facilitar la reutilización de software y la interoperatividad de los componentes.
- DCOM (*Distributed COM*). Extiende las prestaciones ofrecidas por el modelo COM a través de las redes de computadores.
- MTS (*Microsoft Transaction Server*). Servidor de transacciones de Microsoft. Permite la administración de hilos, gestión de transacciones, operación cooperativa de conexiones de bases de datos y seguridad.

Además de integrar los servicios descritos, añade nuevos modelos: de seguridad, de hilos, administración de transacciones, de administración simplificada; y nuevos servicios: cola de componentes (QC), sucesos débilmente vinculados (LCE), operación cooperativa de objetos y administrador de compensación de recursos (CRM).

El modelo COM+ identifica los siguientes elementos:

- Componente COM: módulo software o archivo binario, por ejemplo los controles ActiveX (generalmente una DLL) o los componentes que encapsulan lógica de negocio en una aplicación distribuida.
- Servidor COM: aplicación que ofrece servicios.
- Cliente COM: aplicación que demanda servicios.
- Interfaz COM: conjunto de métodos y propiedades que definen el comportamiento de un componente y que posibilita el acceso a su funcionalidad.

4.2.2.4. .NET Remoting

.NET Remoting proporciona un marco para que los objetos de distintos dominios de aplicaciones, procesos y equipos se puedan comunicar entre sí con compatibilidad en tiempo de ejecución, permitiendo que estas interacciones se realicen de forma transparente. En un sentido más amplio, también podemos entender *.NET Remoting* como una evolución más de los modelos de componentes antes analizados.

Existen tres tipos de objetos que se pueden configurar como objetos de servicios remotos .NET, y entre los que se puede seleccionar el tipo que mejor se adapte a los requisitos de la aplicación:

- **Objetos Single Call**. Los objetos *Single Call* sólo pueden cubrir una solicitud entrante. Este tipo de objetos resulta bastante útil en aquellos escenarios en los que se debe realizar una determinada cantidad de trabajo y, por lo general, no precisan, ni pueden, almacenar información de estado entre las distintas llamadas a métodos. No obstante, se pueden configurar de forma que permitan el equilibrio de la carga.
- **Objetos Singleton**. Los objetos *Singleton* sirven a varios clientes y, por lo tanto, pueden compartir información al almacenar el estado entre las llamadas de los clientes. Resultan bastante útiles cuando los datos se deben compartir obligatoriamente entre los clientes, y en aquellas situaciones en las que se produce un uso significativo de los recursos derivados de la creación y mantenimiento de los objetos.
- **Objetos activados en el cliente** (CAO). Los objetos activados en el cliente son objetos del lado del servidor que se activan cuando el cliente realiza una solicitud. Este método de activación de los objetos de servidor es muy similar al de la clásica activación de una clase asociada «*CoClass*» de COM.

Los dominios de aplicaciones y las aplicaciones .NET se comunican entre sí a través de mensajes. Los denominados *servicios del canal de .NET* facilitan el medio de transporte necesario para establecer estas comunicaciones.

El marco .NET proporciona dos tipos de canales: los *canales HTTP* y los *canales TCP*, aunque los productos de terceros pueden escribir y conectarse a sus propios canales. El canal HTTP utiliza el protocolo SOAP de forma predeterminada para establecer la comunicación, mientras que el canal TCP recurre a la carga binaria.

Los objetos de servicios remotos .NET se pueden alojar en:

- **Un ejecutable administrado**. Los objetos de servicios remotos .NET se pueden alojar en cualquier archivo .NET EXE normal o en un servicio administrado.
- **Internet Information Server** (IIS). De forma predeterminada, estos objetos reciben los mensajes a través del canal HTTP. Para alojar objetos de servicios remotos en IIS se debe crear una raíz virtual, en la que se copiará un archivo `remoting.config`. El ejecutable o la DLL que contiene el objeto remoto se debe colocar en el directorio

`bin`, en el directorio al que señala la raíz de IIS. Se pueden exponer los objetos de servicios remotos .NET como servicios Web.

- **Servicios de componentes .NET**. Los objetos de servicios remotos .NET se pueden alojar en la infraestructura de servicios de componentes .NET para aprovechar los diferentes servicios de COM+, tales como las transacciones, JIT, la agrupación de objetos, etc.

4.2.3. CORBA

CORBA (*Common Object Request Broker Architecture*) es un diseño de middleware que, análogamente a J2EE o a .Net framework, permite que los programas de aplicación se comuniquen unos con otros con independencia de sus lenguajes de programación, sus plataformas hardware y software, las redes sobre las que se comunican, su ubicación y sus implementadores. De hecho, CORBA representa la primera propuesta de middleware aceptado como tal.

Esta arquitectura común para la negociación de solicitudes de objetos fue propuesta por el grupo de administración de objetos OMG. Su principal cometido es trasladar una petición de un objeto cliente —componente CORBA— a una implementación de un objeto, proporcionando los mecanismos por los que los objetos hacen peticiones y reciben respuestas de forma transparente.

Para ello proporciona un modelo de programación sobre bloques básicos arquitectónicos y un nivel de abstracción que permite a las aplicaciones desarrolladas sobre él obtener independencia con respecto a las plataformas sobre las que actúan.

CORBA define los siguientes elementos:

- IDL (*Interface Definition Language*). Lenguaje de definición de interfaces.
- CDR (*CORBA Data Representation*). Representación de datos de CORBA.
- ORB (*Object Request Broker*). Intermediario de petición de objetos.
- IIOP (*Internet Inter ORB Protocol*). Protocolo Inter-ORB para Internet.
- GIOP (*General Inter ORB Protocol*). Protocolo General Inter-ORB.
- RMI (*Remote Method Invocation*). Invocación a métodos remotos.

Las aplicaciones se construyen mediante objetos CORBA, que implementan interfaces definidas en IDL. Los clientes acceden a los

métodos de las interfaces de los objetos CORBA mediante RMI. El componente middleware que da soporte a RMI es ORB.

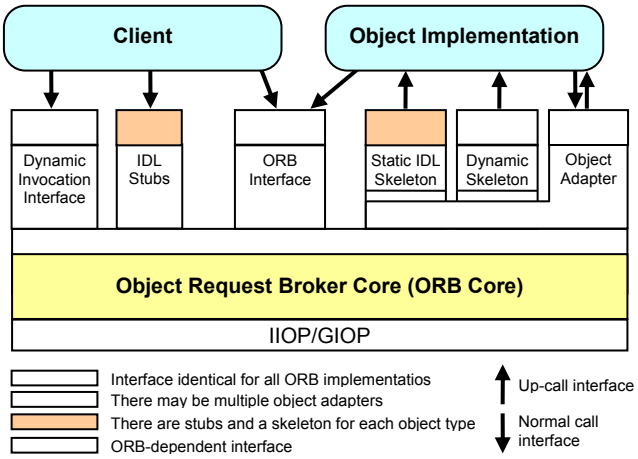

Figura 4.11. Arquitectura CORBA.

Los servicios CORBA proporcionan un equipamiento genérico que puede utilizarse en una gran variedad de aplicaciones. Estos servicios incluyen el servicio de nombres, los servicios de eventos y notificación, el servicio de seguridad, los servicios de transacción y concurrencia y el servicio de comercio.

4.3. INTEGRACIÓN DE TECNOLOGÍAS

Durante estos primeros cuatro capítulos del libro hemos ido desgranando, una tras otra, múltiples tecnologías, herramientas e implementaciones orientadas al desarrollo de grandes aplicaciones distribuidas sobre Internet. El problema es que este tipo de aplicaciones son complejas y, en muchas ocasiones, disponer de un abanico tecnológico tan amplio puede resultar un verdadero quebradero de cabeza a la hora de contestar a preguntas como: qué tecnología es la más adecuada en cada momento, cómo emplearla de la mejor forma y, sobre todo, cómo trabajar con diferentes propuestas de diferentes fabricantes al mismo tiempo.

Una de las mejores opciones para afrontar este tipo de problemas es aplicar soluciones que ya han demostrado ser exitosas en otros supuestos. Este tipo de soluciones se han ido recogiendo desde los años 60 en especificaciones que denominamos «patrones». Existen patrones en muy

diversos ámbitos, como en el diseño, en la optimización o en la organización de las aplicaciones.

En este apartado veremos cómo un patrón de diseño conocido como MVC, aplicado a las diferentes tecnologías que propone J2EE, puede resultar de gran ayuda. Por supuesto, la propuesta es directamente aplicable a otras tecnologías como .NET.

4.3.1. El paradigma MVC

Las tecnologías vistas se enmarcan dentro del desarrollo de aplicaciones Web orientado a objetos y, por lo tanto, deberían responder al *patrón de diseño Modelo-Vista-Controlador* (MVC). Este patrón propone dividir la aplicación en tres partes diferenciadas:

- **El Modelo** o la lógica de negocio de la aplicación. De entre todos los elementos tecnológicos que han ido apareciendo alrededor del modelo Web básico, y que hemos estudiado en capítulos anteriores, el papel de *modelo* queda reservado a los *servidores de aplicación*, gestionados mediante un *marco de trabajo middleware* y basando los desarrollos en el modelo de *componentes software distribuidos* al que estos marcos proporcionan servicios.
- **La Vista** es responsable de gestionar el nivel de presentación, proporcionando una interfaz de usuario altamente desacoplada, tanto de la lógica de negocio como, sobre todo, del cliente. En la práctica, resulta muy aconsejable poder aislar al máximo este apartado del resto de la aplicación, de forma que se pueda dedicar personal especializado en aspectos como el diseño gráfico y de interfaces de usuario más que en el de aplicaciones.
- **El Controlador**. Es el componente que se encarga de manejar la interacción entre la *vista* y el *modelo*, y tiene definido el flujo de la aplicación. Actúa como catalizador necesario para desarrollar las dos partes anteriores. De su elección depende principalmente que se puedan diferenciar nítidamente los distintos roles que deben establecerse para el desarrollo de aplicaciones Web.

Como se ha comentado ya, la lógica de negocio o *Modelo* deberá estar codificada en forma de componentes software, que la encapsulen y la hagan portable y reutilizable entre diferentes sistemas. Sin embargo, la manera en la que repartimos los otros dos papeles es algo abierto a discusión.

Veamos a continuación las distintas posibilidades utilizando como ejemplo de tecnología: servlets por parte de la ejecución de programas, JSP por parte de las páginas activas y Enterprise JavaBean por parte del middleware y de los componentes. En general, la mayor parte de las reflexiones que aquí se realizan son válidas para otros tándems —por ejemplo, el basado en ASP y COM+ o el basado en Liveware y CORBA, por no mencionar todas las posibilidades de interacción entre todos ellos empleando lenguajes de definición normalizados como XML.

4.3.1.1. Aplicación Web basada en JSP

Utilizando únicamente esta tecnología, la lógica de programación se implementa directamente en las páginas JSP. Las páginas JSP harán el papel de controlador y vista a la vez. Si esto es así, los navegadores deberán hacer peticiones directamente a estas páginas, las cuales interactuarán con los componentes (Beans) necesarios e insertarán los resultados obtenidos en formato HTML, el cual volverá al cliente.

Las páginas JSP se encuentran, además, perfectamente integradas con el entorno general del servidor de aplicaciones, pudiendo relacionarse con las otras *piezas* que integran el mismo.

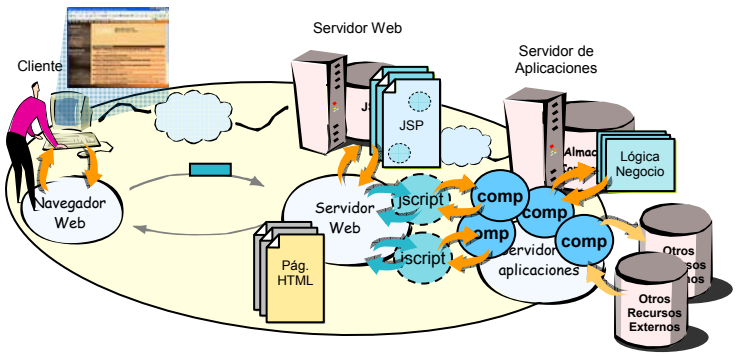

Figura 4.12. Escenario en el que se despliega una aplicación distribuida organizada según un patrón MVC implementado con tecnología JSP.

Es importante recordar que el código fuente (los *scripts*) incrustado en las páginas JSP se extrae y, a partir del mismo, se generarán sus respectivos objetos *servlets,* que es lo que realmente ejecutará el contendor JSP.

4.3.1.2. Aplicación Web basada en Servlets

En este modelo, los servlets hacen el papel de controlador y de vista a la vez, es decir, contienen la lógica de programación y el resultado o presentación final codificados en Java. Esto implica que cada vez que haya una modificación de formato, aunque no la haya de lógica, debemos modificar y recompilar el servlet correspondiente.

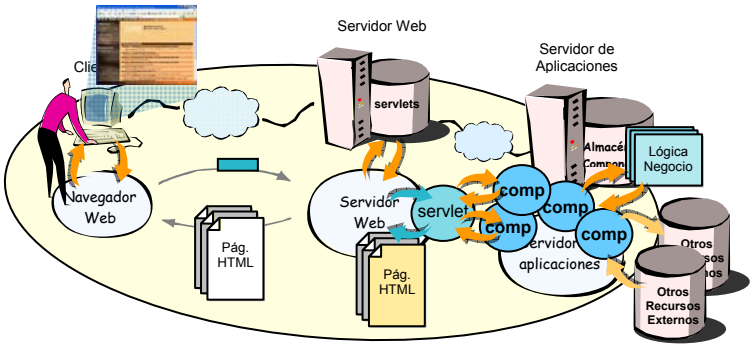

Figura 4.13. Escenario en el que se despliega una aplicación distribuida organizada según un patrón MVC implementado con tecnología *Servlet*.

Aunque aporta un mecanismo muy potente, en la práctica, no facilita la independencia necesaria entre los diferentes papeles de equipo de trabajo que deben confluir en el desarrollo de este tipo de sistemas.

4.3.2. MVC aplicado a J2EE

En este caso, la especificación J2EE fue realizada por SUN para, precisamente, abordar este tipo de problemas inherentes al desarrollo de aplicaciones distribuidas. Por lo tanto, cuenta con todos los ingredientes para desplegar totalmente el patrón MVC. De hecho, la propia SUN, consciente de la complejidad que entraña la utilización de su plataforma, fue la primera en facilitar a sus usuarios su propia versión sobre cómo aplicar el patrón MVC con J2EE. Esta especificación se conoce como «el modelo 2».

Según el *modelo 2 de SUN*, la lógica de programación irá ubicada en servlets, que posteriormente invocan a páginas JSP. Podemos colocar la lógica de programación en los servlets, de forma que los clientes les

invoquen y éstos, a su vez, a páginas JSP, las cuales accederán a la información que necesiten sólo para presentar resultados.

Serán los servlets los que interactuarán con los JavaBeans con el fin de realizar cualquier cálculo o acceder a la lógica de negocio. Los servlets podrán además crear Beans donde almacenar los resultados obtenidos.

Posteriormente, las páginas JSP extraerán la información requerida y almacenada en estos Beans, con el fin de insertarla en su HTML. El resultado final, tras mezclar el contenido dinámico con el estático, se envía al cliente o navegador.

Las páginas JSP constituyen una tecnología mediante la cual podemos presentar páginas con contenido dinámico en base a etiquetas (*tags*) especiales embebidas en el código HTML de la página. Mediante estas etiquetas podremos incrustar código Java para ejecutar lógica de aplicación directamente o bien acceder a un componente externo que realice esta tarea por él, devolviéndole posteriormente un resultado.

Al actuar los servlets como controladores, las páginas JSP, al margen del HTML normal de la página, deben llevar sólo *tags* especiales para su relación con los JavaBeans.

Usando los llamados «*scriplets*» podremos introducir código Java en nuestras páginas JSP. Pero no debemos abusar de esta funcionalidad, ya que cuanto más lo hagamos, más dificultoso será separar los diferentes roles o perfiles en un equipo de desarrollo. En la medida de lo posible, la lógica de programación deberá ir en los servlets, la lógica de negocio en los JavaBeans y la presentación o interfaz gráfica en páginas JSP.

Este último enfoque es el método que juzgamos más idóneo, ya que separa la vista o presentación (en páginas JSP) de la lógica de programación (en servlets) y aporta la indudable ventaja de que podemos separar el contenido de presentación o estático del contenido dinámico, estando la lógica de negocio en componentes o Beans externos a la página.

Según este esquema, en la figura 4.14 podemos apreciar una representación gráfica de la política de trabajo propuesta en este apartado. El cliente interactúa con los servlets, en los que va ha estar albergada la lógica y el flujo de programación. Desde ellos se accede a las clases que encapsulan toda la lógica de negocio (*Business Objects*), que son independientes de la filosofía Web.

Los resultados que se obtengan se almacenan en Beans (*View Objects*) y se presentan a través de páginas JSP, las cuales tienen acceso a los mismos para obtener la parte dinámica. Si la presentación es estática, se presentan simples páginas HTML.

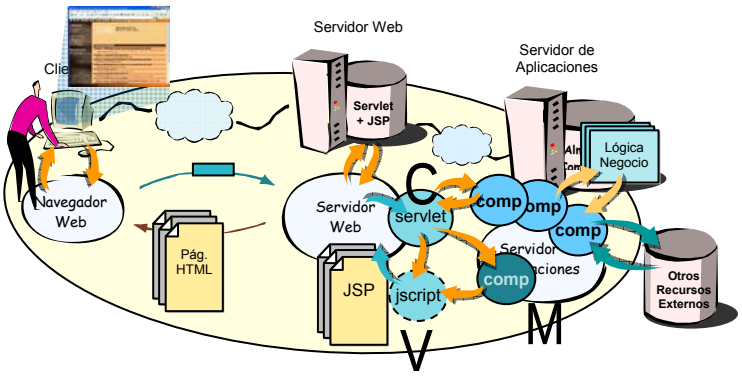

Figura 4.14. Escenario en el que se despliega una aplicación distribuida organizada según un *modelo 2* propuesto por SUN en el que se identifica el (M)odelo, la (V)ista y el (C)ontrolador.

Si somos capaces de separar en la medida de lo posible estos dos mundos —presentación y lógica—, estaremos diferenciando también entre el papel del personal dedicado al diseño y a la usabilidad, de aquél dedicado a plasmar el modelo de negocio, lo que redundará en su especialización, pudiendo cada uno de ellos emplear para su cometido las mejores herramientas en cada uno de los campos.

4.3.3. Escenario de desarrollo

En la figura 4.15 se presenta un posible escenario de desarrollo, más o menos complejo, en el que confluyen la mayor parte de los elementos abordados en esta lección.

Por parte de los clientes, podemos observar diferentes tipos de dispositivo y mecanismos de interconexión a la Red desde PCs convencionales, hasta PDAs o teléfonos móviles. Por parte del servidor Web, se plantea un sencillo cluster (posiblemente soportando balanceo de carga) que representará el punto de entrada a la aplicación; por supuesto, apoyándose desde el punto de vista operativo en los dispositivos típicos de red y seguridad como *routers*, *proxys* o *firewalls*. En lo que respecta a los servidores de aplicaciones, se han separado los nodos de ejecución de la aplicación o de los componentes software de aquellos nodos que pueden servir como almacén de componentes. Finalmente, se proporciona un entorno de almacenamiento tipo NAS (*Network Attached Storage*) o SAN (*Storage Area Networks*), que puede corresponder tanto a aplicaciones y

sistemas heredados como a almacenes de persistencia de los propios componentes.

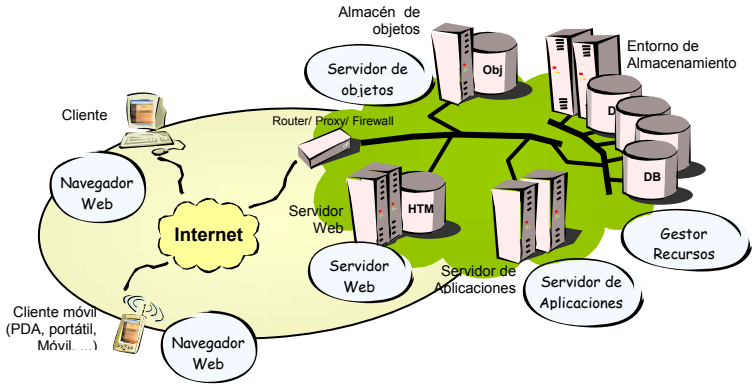

Figura 4.15. Escenario general en el que se recogen los principales componentes físicos y lógicos que proporcionan soporte a una aplicación distribuida sobre Internet.

4.4. CASO DE USO: JBOSS

JBoss es un servidor de aplicaciones J2EE de código abierto implementado en Java puro. Al estar basado en Java, JBoss puede ser utilizado en cualquier sistema operativo que soporte la máquina virtual de java. Los principales desarrolladores trabajan para una empresa de servicios, JBoss Inc., adquirida por Red Hat en abril del 2006, fundada por *Marc Fleury*, el creador de la primera versión de JBoss. El proyecto está apoyado por una red mundial de colaboradores. Los ingresos de la empresa están basados en un modelo de negocio de servicios.

JBoss AS fue el primer servidor de aplicaciones de código abierto, preparado para la producción y certificado J2EE 1.4, disponible en el mercado, ofreciendo una plataforma de alto rendimiento para aplicaciones de negocio electrónico (actualmente se está desarrollando la versión para J2EE 5). Combinando una arquitectura orientada a servicios revolucionaria con una licencia de código abierto, JBoss AS puede ser descargado, utilizado, incrustado, y distribuido sin restricciones por la licencia. Por este motivo es la plataforma más popular de middleware para desarrolladores, vendedores independientes de software y, también, para grandes empresas.

Las características destacadas de JBoss incluyen:

- Producto de licencia de código abierto sin coste adicional.