TECNOLOGÍA Y HERRAMIENTAS

Ricardo García Mataix
Técnico de Sistemas
Hospital La Fe de Valencia

INTRODUCCIÓN

En la anterior edición de los Informes SEIS se presentaba una visión general del panorama actual de las tecnologías de la información y su aplicabilidad en el ámbito de los sistemas de información en los entornos sanitarios.

Los sistemas de información en el ámbito de la salud y en especial en los hospitales se caracterizan por su inmensa complejidad, tanto por su extensión como por tener que lidiar con modelos de datos muy ricos y cambiantes. Si deseamos adentrarnos en el mundo del diseño y el desarrollo de software en este entorno, parece necesario acometer tal problemática utilizando las últimas tecnologías en cuanto a ingeniería del software se refiere.

A finales del año 1999, la división Java de Sun Microsystems publica la primera versión de J2EE, una especificación para el desarrollo de aplicaciones distribuidas basada en el lenguaje Java. Desde entonces, J2EE ha ido evolucionando hasta su última versión, la 1.4, y se ha convertido en uno de los grandes paradigmas en el mundo actual de la informática profesional.

Pretendemos aquí dar una visión general del estándar J2EE, explicando sus componentes principales, los servicios que ofrece y analizar su posible aplicabilidad en el entorno hospitalario.

El artículo, sin embargo, no pretende ser únicamente una exposición aséptica e impersonal de definiciones, nuestra intención es también la de argumentar y tomar partido por algunas de las opciones que se nos presentan, invitando así a la discusión.

Finalmente, describimos a grandes rasgos una posible arquitectura que se podría adoptar hoy para el diseño realista de un gran sistema de información en el entorno hospitalario, viendo de una forma muy general los elementos que la integrarían.

¿POR QUÉ J2EE?

J2EE es un estándar de facto basado en el lenguaje de programación Java en el que han participado las mayores empresas (a excepción de Microsoft) del mundo de la informática. Si bien, la dirección la lleva Sun, otros gigantes como IBM, Oracle o BEA participan directamente en él.

La característica más relevante es que J2EE es una especificación, es decir, no es un producto acabado. De esta especificación existen muchas implementaciones. Esto significa, en primer lugar, que vamos a poder elegir entre muchas opciones, beneficiándonos de la competitividad entre ellas. Además, la gran mayoría de estas implementaciones funcionan sobre cualquier plataforma hardware y sistema operativo. Gracias a ello, la adquisición de un servidor de aplicaciones J2EE no nos va a condicionar la elección de los servidores físicos y del software de base. Una implementación de J2EE es lo que se denomina técnicamente un servidor de aplicaciones J2EE. Existen multitud de servidores certificados como servidores J2EE, entre los que hay que destacar: Weblogic de BEA, Oracle 10g, Websphere de IBM, Sun Java System Application Server, JBoss (open source y gratuito), JRun de Macromedia, Borland Application Server, Pramati y muchos más.

Si bien todos estos productos implementan J2EE, ello no significa que sean idénticos. Algunos lo hacen mejor que otros y además tratan de distinguirse de sus competidores ofreciendo características que no contempla la especificación. La elección del servidor de aplicaciones que vaya a alojar las aplicaciones de un hospital es una decisión fundamental y estratégica. En principio, como software de base que es, el servidor de aplicaciones no debería dar muchos problemas. Sin embargo, esta ausencia total de fallos nunca se cumple del todo. De la misma manera que el sistema operativo o la base de datos puede fallar, también puede fallar el servidor de aplicaciones. Es, pues, vital disponer de un buen mantenimiento de esta pieza de software, un mantenimiento muy especializado que, seguramente, nos tendrá que ofrecer la misma casa que nos lo vendió o alguna otra empresa de servicios. Tal pieza de software supone, además, una figura nueva en la organización del servicio de informática, el administrador del servidor de aplicaciones.

Optar por J2EE frente a otras alternativas –como .NET, un producto propietario de Microsoft– sólo significa haber elegido un lenguaje y un gran marco para el desarrollo de nuestras aplicaciones. Sin embargo, aún se tienen que tomar muchas decisiones. El hecho es que J2EE ofrece varios grandes paradigmas de desarrollo, pues pretende ofrecer una solución a casi todos los problemas de software imaginables. De entre toda la maraña de siglas que conforman la especificación, tendremos que elegir qué concreción de J2EE se adapta mejor a nuestro problema concreto. En este artículo vamos a suponer que tratamos de diseñar un sistema de información que dé respuesta a las necesidades de un hospital de nueva creación. En algunos momentos comentaremos también como se lidiaría con la situación de convivir con aplicaciones ya existentes en nuestra organización (legacy).

UNA VISIÓN GENERAL DE J2EE

Java 2 Enterprise Edition específica una plataforma para la ejecución de grandes aplicaciones distribuidas.

J2EE es muy extenso y complejo. Su complejidad proviene del hecho de su pretensión de ser una plataforma de software total. La última versión es J2EE 1.4 y tiene 250 páginas. Sin embargo, la especificación hace referencia a otras muchas especificaciones que presupone como conocidas (EJB 2.1 650 páginas, Servlet 2.4 330 páginas, JSP 2.0 478 páginas, JMS 1.1 140 páginas y muchas otras especificaciones más) y presupone la existencia y conocimiento de todas las APIs nucleares de Java, las descritas en su hermana menor J2SE (Java 2 Standard Edition). Además, las especificaciones no están escritas con el objetivo de explicar al profano los temas que trata, se trata más bien de una exposición dogmática sobre cómo deben comportarse los productos que pretendan implementar los componentes indicados. Unos buenos manuales explicando y ejemplificando las materias específicas ocupan aún mucho más espacio y mucho tiempo de estudio.

Otro factor de complejidad viene por el hecho de que para poner en práctica una aplicación J2EE nos hará falta una gran cantidad de software que hay que acoplar para que todo funcione. Con todo ello no quiero invitar al desaliento, sino apuntar ya la idea sobre la necesidad de disponer en la propia organización de personal que tenga un cierto conocimiento del tema. En caso contrario, fácilmente podemos ser seducidos por arquitecturas de aplicación que sirvan más a los intereses de los proveedores que a los clientes. Se hace muy difícil rebatir decisiones de diseño que pueden resultar estratégicas, si la organización cliente no posee el más mínimo "know-how" sobre estos temas. "Outsourcing" total puede ser muy arriesgado.

A continuación, proponemos hacer inteligible los aspectos más relevantes de la especificación, más como un ejercicio de traducción que como una exposición lineal y descriptiva.

¿Cómo subdividir una gran aplicación conceptualmente? El gran meta-patrón de diseño ideado para ello es el patrón modelo-vista-controlador (MVC). Existen ligeramente distintas interpretaciones de este patrón en función del contexto en el que se aplique. Vamos a suponer, a modo de ejemplo, que deseamos desarrollar un gran HIS. La idea fundamental es modelar por un lado los procesos y entidades consustanciales a nuestra organización. Hospitalizar, dar de alta, pedir una analítica serían ejemplos de este tipo de procesos: representan la actividad que realiza todo hospital, su razón de ser. Ejemplos de entidades de nuestro sistema serían: Paciente, Facultativo, pero también Determinación o Diagnóstico. La idea es que esta parte de

la aplicación es la nuclear y, en principio, no ha de estar sujeta a cambios con demasiada frecuencia. Daremos otra vuelta alrededor de esta idea más adelante.

Los Enterprise Java Beans

El componente software que ofrece J2EE para el modelado de procesos y entidades son los Enterprise Java Beans (EJBs). Existen 3 tipos de EJBs: beans de sesión, para modelar procesos síncronos, message driven beans para procesos asíncronos y entity beans para representar entidades. Los EJBs son componentes software que se ejecutan dentro de un contenedor específico para ellos. Este contenedor es uno de los constructos que nos ha de ofrecer el servidor de aplicaciones que hayamos elegido. Este contenedor es quien consigue que los EJBs sean accesibles de forma remota, además de ofrecerles toda una serie de servicios también explicitados en J2EE. De esta forma, nuestros procesos y entidades formarían un cuerpo común ejecutándose en un servidor lógico y físico y serían accesibles de forma remota por cualquier tipo de clientes como explicamos a continuación.

J2EE declara que los EJBs (nuestros procesos y entidades) han de ser accesibles de forma remota a través del protocolo IIOP (o JRMP, uno propio de Java más eficiente cuando tanto el cliente como el servidor son clases Java), y también de forma local, mediante invocación directa de sus métodos a través de cualquier clase Java que se ejecute con ellos dentro del mismo servidor de aplicaciones y en el mismo contexto.

Ello significa que el corazón de nuestro sistema puede ser conectado directamente a cualquier aplicación de escritorio que entienda el protocolo IIOP, ello incluye, como era lógico esperar, a los clientes Java.

Los clientes Java posibilitan la creación de sofisticadas aplicaciones de escritorio que se ejecutan en los PCs u otros dispositivos de que disponga el cliente. Sin embargo, tienen un gran problema, que viene por el hecho de que el protocolo IIOP (idem para JRMP) no pasa fácilmente por los firewalls (cortafuegos) que cualquier organización seria tendrá entre Internet exterior y su propia Intranet. Veremos más adelante qué implicaciones tiene este hecho.

Existen dos tipos de beans de sesión: sin estado y con estado. Los segundos se asignan directamente a un cliente que obtenga una referencia a él y pueden mantener información de estado entre distintas invocaciones de sus métodos. La funcionalidad de los primeros es muy parecida a la de un servidor Web, acepta peticiones, contesta, pero no mantiene memoria entre una invocación y otra.

Los beans de mensajería sirven para conectar un cliente con un servidor de mensajería JMS (Java Message Service), tanto para la invocación asíncrona de méto-

dos, como para registrarse como interesados en la recepción de mensajes según su pertenencia a un Topic, por ejemplo algún tipo de evento interesante para él.

Los beans entidad representan entidades de nuestro sistema. Existen dos tipos de beans de entidad en función de la manera en que sus atributos se persisten, es decir se pasan a un almacenamiento permante de datos, generalmente una base de datos relacional: entity beans BMP (Bean managed persistence) y entity beans CMP (Container managed persistence). En los primeros, es el propio desarrollador de los beans el que ha de escribir la lógica de recuperación y actualización de datos de la base de datos a memoria y a la inversa (básicamente, escribiendo código Java que utilice el API JDBC contra el driver de la base de datos que se nos haya suministrado). En el segundo caso, se especifica cómo queremos mapear los atributos de nuestros beans a qué atributos de las tablas de nuestras bases de datos. Ello se realiza utilizando un lenguaje XML ad-hoc. El contenedor de EJBs es el que tras leer esta especificación habrá de generar el código necesario para conseguir la persistencia.

El contenedor Web

¿Qué ocurre si queremos conectarnos con el contenedor de EJBs desde el exterior, típicamente a través de un navegador? J2EE ofrece lo que denomina un contenedor WEB que sirve, entre otras cosas, para solucionar este caso. Un contenedor Web alberga servlets y páginas JSP. Acepta peticiones http de un navegador, las transforma en objetos Java que les pasa a los servlets y/o páginas JSP. Éstas a su vez le proporcionan objetos Java al contenedor y éste vuelve a tranformarlos en mensajes http de respuesta que el servidor web redireccionará al cliente.

Para el programador, un servlet no es más que una clase Java normal al que se le pasan dos objetos especiales como parámetros de sus métodos. Un objeto encapsula la petición http y el otro la respuesta http. El programador, en principio, tan sólo tiene que sobrescribir estos métodos y escribir la funcionalidad deseada. Una vez haya conseguido esto, despliega el servlet en el contenedor Web, indicando entre otras cosas la URL con la que se desea que el cliente pueda acceder a él.

En muchos casos lo que se desea devolver al cliente será una página HTML cuyo contenido se ha construido dinámicamente. Para no tener que generar la página HTML desde dentro de los servlets, J2EE ofrece las Java Server Pages (JSP). Éstas, aparentemente, son páginas HTML en las que se permite incrustar directamente la escritura de código Java o la referencia a librerías Java. Al invocar la JSP, el contenedor WEB compondrá la página HTML resultante tras ejecutar el código indicado en los lugares indicados.

Además, cualquier cliente puede acceder a los métodos de los EJBs de sesión sin estado viéndolos como servicios Web (a partir de J2EE 1.4).

Por supuesto, también podemos desarrollar aplicaciones Web sin necesidad de utilizar los EJBs en absoluto. De hecho muchas organizaciones han adoptado este tipo de arquitectura, utilizando navegadores como clientes. Para este tipo de arquitectura, sin embargo, no se hace completamente necesaria la adquisición de un servidor de aplicaciones, pues existen contenedores Web (capaces de albergar servlets y JSPs) gratuitos y que funcionan razonablemente bien, como es el caso de Apache-Tomcat.

Finalmente, J2EE ofrece la posibilidad de la utilización de applets. Se trata, básicamente, de programas Java que se ejecutan desde y dentro de un navegador con el propósito de ofrecerles funcionalidades que estos no incluyen. Veremos brevemente algunas características de los applets más adelante.

Servicios J2EE

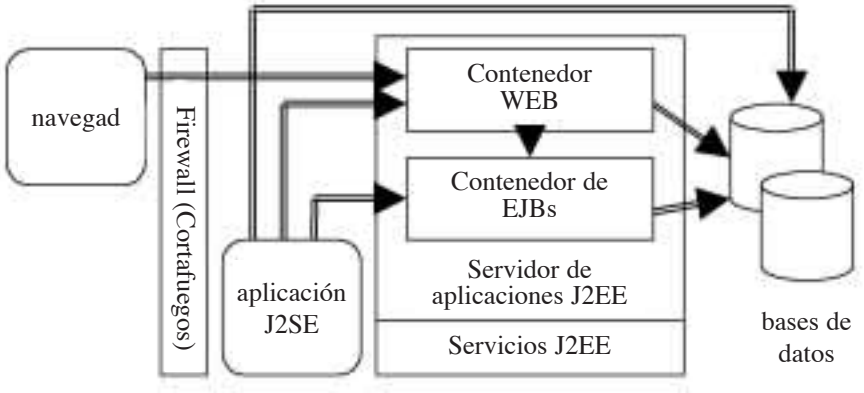
J2EE, además, específica una serie de APIs, funcionalidades, que cualquier servidor de aplicaciones certificado debe ofrecer a cualquier componente que se ejecute en cualquiera de sus contenedores. No vamos a agotar al lector con una descripción pormenorizada de estos servicios. Destacamos simplemente: APIs para todo tipo de trabajo con ficheros XML, JDBC (Java Database Connectivity) para el acceso a bases de datos relacionales, JCA (Java Connector Architecture) para conectarse con sistemas extraños a J2EE pero que aceptan una serie de contratos con la plataforma J2EE, soporte para transacciones distribuidas y otros muchos más.

Hay que destacar que J2EE no especifica un lugar donde persistir el modelo del sistema, algo que, por otra parte, resultará absolutamente necesario. Como hemos comentado, J2EE incorpora una API propia para la conexión con bases de datos relacionales (JDBC) y otra para conectarse a otro tipo de arquitecturas (JCA), pero no presupone la existencia de una base de datos u otra. Habrán de ser los fabricantes de bases de datos (Oracle, IBM-Informix, Sybase, PostgreSQL, MySQL, etc.) los que nos ofrezcan sus drivers JDBC para conectarnos a sus bases de datos o sus conectores JCA para conectarnos a sus sistemas.

Ello no nos ha de sorprender, pues todas las grandes compañías de bases de datos intervienen directamente en la especificación y no tienen el más mínimo interés en perder cuota de negocio en este sentido.

Así pues, el mapa de posibilidades J2EE sería el siguiente:

Figura 1



Como vemos, las posibilidades para la construcción de aplicaciones son muy grandes.

Como hemos apuntado anteriormente, ello no significa que tengamos que utilizar todos los componentes de J2EE para construir cualquier aplicación, sino saber cuales elegir y por qué lo estamos haciendo. La postura que vamos a defender para la construcción de este hipotético HIS no es, quizá, la más extendida, por ello trataré de razonarla y argumentarla lo más claramente posible. Para ello vamos a explicar brevemente la evolución histórica de la arquitectura de aplicaciones. Más adelante describiremos el contexto en el que nos estamos moviendo, un hospital de tamaño medio o grande y veremos como aplicar las últimas tecnología a este entorno.

EL CAMINO HASTA J2EE

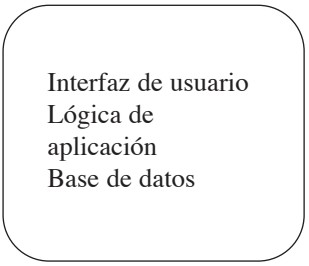
La evolución de la arquitectura de las aplicaciones en los últimos años ha sido vertiginosa. Por otro lado, la situación actual tampoco parece estabilizarse demasiado. Muchos son los que piensan que se ha complicado en exceso. Es cierto, pero es que ahora podemos abordar el desarrollo de sistemas cuya complejidad los convertía en intratables, algo que no siempre se supo reconocer y que abocó al fracaso a gran cantidad de proyectos excesivamente ambiciosos.

Arquitectura monolítica

Desgraciadamente, aún quedan muchos aplicaciones hospitalarias diseñadas en base una arquitectura de una capa, la más simple posible. En este tipo de aplica-

ciones, toda la aplicación se ejecuta en una única máquina. Las desventajas de este tipo de arquitectura son evidentes. Es más difícil destacar alguna ventaja sobre otro tipo de arquitecturas. Una ventaja podría ser que la cualificación del personal informático para desarrollar y mantener este tipo de entornos no suele ser muy grande.

Figura 2



El mantenimiento y posibilidades de crecimiento de este tipo de arquitecturas se hace cada vez más difícil, dando lugar a situaciones insostenibles. Algunos HIS diseñados a finales de los 80 o principios de los 90 han seguido este modelo. Por otro lado, al representar HIS el núcleo de los sistemas de información que funcionan en un hospital, su sustitución por sistemas más modernos se hace tan difícil, que en algunos casos se ha preferido mantenerlos por no poder soportar la organización el coste de la migración a sistemas más avanzados tecnológicamente.

Arquitecturas cliente servidor

Gran parte de las aplicaciones desarrolladas durante los años 90 se han diseñado en base a la conocida arquitectura cliente servidor.

Disponer la base de datos en una segunda máquina, un servidor, supuso una gran revolución en su época, pues posibilitó el desarrollo de clientes que se ejecutaban en los PCs. Así, las aplicaciones devinieron visualmente más agradables e intuitivas, muy similares a las aplicaciones de escritorio (procesadores de texto, hojas de cálculo) que empezaban a ser muy utilizadas en aquella época. Muchos sistemas departamentales se desarrollaron de esta manera. Algunos hospitales, gracias a haberse rezagado en su grado de informatización, consiguieron un HIS cliente servidor, distinguiéndose orgullosamente de sus homólogos monolíticos.

Sin embargo, la arquitectura cliente servidor se vio en grandes dificultades en cuanto la aplicación desarrollada fue creciendo en funcionalidad y el número de usuarios que había de utilizarla iba en aumento. Por un lado, toda la funcionalidad

de la aplicación sigue fuertemente acoplada, por lo que resulta muy fácil escribir rápidamente grandes cantidades de código pero en base a un mal diseño.

Existe un importante principio en el desarrollo de software, difícil de explicar y de comprender, pero que ha tenido una influencia decisiva en el fracaso de muchos grandes proyectos de software. El problema viene del hecho de que para resolver un problema concreto de software la solución más rápida, y por lo tanto, la que suele ser la preferida por los desarrolladores -y exigida por los analistas, que a su vez había sido la pautada por los directores de proyecto, pues era lo que habían vendido a los que habían comprado y pronto reclamado el software en cuestión- suele ser la peor solución, en el sentido que será aquella que sumada a otras rápidas y malas soluciones a problemas concretos irán construyendo un sistema cada vez más acoplado ("más chapuza, para entendernos") y en el que, finalmente, el intento de mejorar o simplemente cambiar algo se convierte, cada vez más, en una tarea titánica, imposible de asumir por el equipo de desarrollo. Se trata del clásico problema de falta de diseño, un corolario del principio de falta de visión a medio o largo plazo. Este problema es bien conocido, se ha formulado en multitud de ocasiones, pero no por ello se ha dejado de ignorar una y otra vez.

J2EE no evita estos problemas, pero si que establece ya una clara división entre componentes para el desarrollo de interfaz de usuario y componentes para la implementación del núcleo del sistema.

Un problema clásico de las aplicaciones cliente servidor es el mantenimiento de los puestos cliente. Sin embargo, este problema se ha visto superado en parte, gracias a la posibilidad de control remoto de puestos cliente y a la existencia de mecanismos de actualización automática de versiones. Las aplicaciones de escritorio escritas en Java ofrecen esta última posibilidad gracias a la tecnología JavaWebStart que comentaremos más adelante.

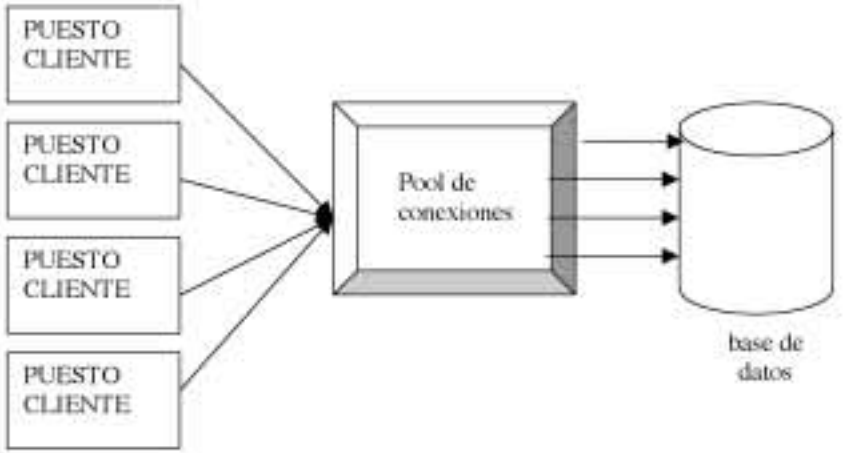
Otro problema de las aplicaciones cliente servidor viene por el hecho de que cada puesto cliente necesita una conexión a la base de datos. Las conexiones cuestan dinero y además consumen grandes recursos de red. Abrir y cerrar las conexiones entre los clientes y el servidor de la base de datos es un proceso extraordinariamente costoso, independientemente de las mejoras en las CPUs, que enlentece la aplicación de forma inaceptable para los usuarios. El sistema que se ha ideado para resolver este problema es el pool de conexiones, que pasamos a explicar.

En algún lugar intermedio entre los puestos cliente y el servidor de la base de datos se abre un número limitado de conexiones. Cada vez que un puesto cliente necesita utilizar una conexión, se la pide al pool de conexiones, éste busca una

conexión libre ya abierta, la marca como ocupada y se la devuelve al cliente. Cuando el cliente ya no necesita la conexión, se lo indica al pool que la marca de nuevo como libre sin cerrar la conexión.

La figura siguiente describe el carácter de concentrador de conexiones que tiene el pool de conexiones a la base de datos.

Figura 3



Arquitecturas multicapa

Algo fundamental en la figura anterior es que el pool de conexiones ha de ser una pieza de software que se ejecuta en una máquina distinta a la de los puestos cliente (la base de datos y el pool pueden, sin embargo, colocarse en el mismo servidor. Vemos así como, de forma natural, aparece la arquitectura distribuida en las aplicaciones: Nos hemos visto forzados a distribuir el diseño de la aplicación entre varios ordenadores. El siguiente paso en esta misma dirección es más conceptual: ¿Si tenemos este servidor central y esta pequeña aplicación que gestiona el pool de conexiones, por qué no ampliar este software central para que ejecute aquellas partes de la aplicación nucleares y dejar en los puestos cliente aquellas otras partes de la aplicación encargadas de la interacción con el usuario final? Esta es la idea del patrón MVC a la hora de particionar y distribuir las aplicaciones que habíamos insinuado anteriormente. Esta distribución de la aplicación en varias capas lógicas y físicas no sólo trae ventajas, también plantea retos importantes. ¿Cómo se comunican los puestos clientes con el servidor, cómo resolvemos la concurrencia de los

accesos a la parte central, quién gestiona el ciclo de vida de esta parte central, qué pasa si el servidor se cae, cómo conseguimos actualizar simultáneamente aquellas partes de la aplicación que sean comunes en los puestos cliente y en el servidor? Esta problemática, o al menos gran parte de ella, es la que pretende resolver el servidor de aplicaciones J2EE. La filosofía consiste en la oferta de toda esta clase de servicios por parte del servidor a los componentes que se ejecutan dentro de sus contenedores, liberando al programador de enfrentarse a este tipo de problemas, más técnicos, más parecidos a programación de sistemas o de bajo nivel.

El resultado de esta idea es que el servidor de aplicaciones le ofrece al programador la posibilidad de que se encargue de resolver los problemas específicos de su sistema particular, un hospital en nuestro caso y que no tenga que dedicarse a una problemática que no es suya y que, además de ser más complicada, es también más genérica.

La aparición de las aplicaciones Web

Con todo lo dicho, el factor más decisivo para explicar y justificar la existencia de J2EE es la aparición de Internet y de las aplicaciones Web. La posibilidad de que clientes de cualquier parte del mundo pudieran ejecutar aplicaciones abrió posibilidades inmensas a finales de los años 90. Como vimos, J2EE ofrece la posibilidad de diseñar aplicaciones Web con o sin la utilización de los sofisticados EJBs.

Sin embargo, Internet no se concibió para la ejecución de aplicaciones remotas, sino como una red muy grande, para poder intercambiar información más bien estática. La arquitectura Web tiene ciertas ventajas:

1) La más evidente es que ya no se hace necesario actualizar las aplicaciones cliente, pues éstas, simplemente no existen. En principio, sólo se requiere un navegador. Veremos que esto no es cierto del todo, pues el propio navegador es ahora la aplicación cliente.

2) El protocolo http es sencillo, fácil de entender y bien estandarizado.

3) Toda la infraestructura software para el desarrollo de aplicaciones Web sin EJBs se puede adquirir de forma gratuita.

4) Los clientes de la aplicación pueden estar ubicados en cualquier parte del mundo

Es curioso, sin embargo, observar que ninguna de estas ventajas incide directamente en una mejora de las aplicaciones desde la perspectiva del usuario final en un entorno hospitalario.

Las aplicaciones Web tienen, por otro lado, muchos inconvenientes:

1) Los navegadores no se concibieron como plataforma de ejecución de aplicaciones. Su integración con los periféricos es muy complicada. Las aplicaciones de escritorio son, por su propia naturaleza, mucho más ricas en el sentido de que posibilitan la creación de una interfaz de usuario mucho más rica e intuitiva. Hay que admitir desde el principio que aplicaciones de escritorio ricas e intuitivas son, quizá, más costosas de desarrollar. Las ventajas son para el usuario final, los inconvenientes para el equipo de desarrollo. Hay que preguntarse para qué y para quien se desarrollan las aplicaciones.

2) En las aplicaciones Web siempre han de viajar la interfaz de usuario más los datos de aplicación, y ello cada vez que se ha de generar una página nueva. Esto significa que vamos a necesitar más ancho de banda en nuestra red.

3) Siempre que se desee actualizar la información en el cliente habrá de ser a instancias del propio cliente, es decir, la parte servidora de la aplicación no tiene forma de comunicarse con el cliente para suministrale información que puede ser de su interés.

4) Los navegadores no tienen la posibilidad de demarcar transacciones, ni siquiera de comunicarse directamente con la base de datos, algo muy recomendable en muchas situaciones.

Para enriquecer la funcionalidad de un navegador se han ideado varias tecnologías. Una de ellas son los applets Java. Se trata, básicamente, de programas J2SE que se referencian desde el código html. El navegador, al "parsear" el código html descarga desde el servidor las clases que conforman el applet y las ejecuta dentro del contexto del navegador. Los applets Java fueron decisivos para el gran éxito de Java, pues gracias a ellos se pudieron ver las primeras animaciones dentro de las -hasta entonces- estáticas páginas Web. El principal problema de los applets es su lentitud, por lo que han perdido mucho terreno a favor de otras metodologías.

La tecnología más utilizada actualmente para enriquecer los clientes en aplicaciones Web es dinamizarlos mediante el uso de un lenguaje específico para ello y que se incrusta dentro de las páginas HTML. De entre todos ellos, Javascript es el que definitivamente ha ganado esta batalla. Javascript, por cierto, no tiene nada que ver con el lenguaje Java. El problema con esta tecnología viene del hecho de que la forma de llevar a cabo la mencionada interpretación no es estándar entre los navegadores. El programador de Javascript ha de estar al tanto del navegador en el que se vaya a ejecutar el programa. El resultado es un código absolutamente ilegible y con un gran coste de conocimiento de todas las particularidades de cada uno

de los navegadores y de cada una de sus versiones. El desarrollo Web requiere, de esta manera, conocimientos muy heterogéneos: navegadores, html, Javascript, servidor Web, servlets, JSP. La publicitada sencillez de las aplicaciones Web suele ser una quimera cuando se desea desarrollar una aplicación no trivial.

Que no se me entienda mal; en muchos casos, las aplicaciones Web son sencillamente las únicas posibles. Lo que no está tan claro es que sean automáticamente las mejores en cualquier situación. A las empresas de software, sin embargo, les interesa mucho tener un único marco de desarrollo. Por este motivo, todas ellas abogan por la opción Web, argumentando algo así como que "es lo más moderno". Como pudimos oír recientemente en un congreso: "Nuestra aplicación, por supuesto, es Web".

Curiosamente, los analistas, diseñadores y programadores de software no utilizan aplicaciones Web para su trabajo diario. No he visto nunca que nadie se plantee implementar un entorno de desarrollo o una herramienta CASE o un procesador de textos o una hoja de cálculo como una aplicación Web. Utilizamos la Web para acceder al correo, leer documentación, descargarnos software, pero los informáticos no utilizamos prácticamente nunca la Web como plataforma de ejecución de aplicaciones.

En cualquier caso, antes de precipitarnos en la toma de decisiones, vamos a ver las características externas de las aplicaciones del entorno que nos ocupa.

EL ENTORNO INFORMÁTICO HOSPITALARIO

Un hospital de gran tamaño, entendido como un sistema de información, representa, sin ningún lugar a dudas, uno de los sistemas más difíciles de diseñar que existen. La exigencia, además, de integrar el hospital dentro del área de salud, de interoperar con otros hospitales y, al menos en el caso de los hospitales públicos, de ser considerados como una pieza de un sistema de información aún más amplio no hace más que aumentar, si cabe, la complejidad del Sistema de Información.

Todos los usuarios han de estar sujetos a las medidas de seguridad de nivel alto que establece la Ley Organica de Protección de Datos de carácter personal que establece el Real Decreto 994/1999. Los sistemas, pues, han de auditar los accesos de los usuarios a los registros de muchas tablas de la aplicación.

Muchas partes de un sistema de información hospitalario han de funcionar las 24 horas del día, los 365 días del año. El impacto que puede producir una caída del sistema de información de un hospital puede considerarse muy grave. Los planes de emergencia en algunos casos no existen, en otros se quedan obsoletos conforme pasa el tiempo. Así, también el hardware que ha de albergar el software hospitala-

rio debe ser muy sofisticado para conseguir alta disponibilidad: clusters de servidores, redundancia de hardware, etc

Además, resulta bastante evidente que en un gran hospital han de convivir distintos sistemas de información, muy heterogéneos, diseñados por diferentes fabricantes. Esta situación es muy difícil que vaya a cambiar en un futuro, pues muchos de ellos son muy especializados (electromedicina, PACS, laboratorios, etc.) y parece imposible concebir un único software que dé respuesta a toda la casuística de un hospital.

Por otro lado, el soporte informático para la práctica asistencial diaria también puede considerarse bastante sofisticado: acceso a la historia clínica, prescripción farmacéutica, edición de informes clínicos, etc.

Existe, además la necesidad de ofrecer una interfaz intuitiva y fácil de manejar que oculte, en ocasiones, algoritmos muy complicados (traslados de cama, citación automática, citación múltiple, reprogramación masiva de citas), muy pesadas (estadísticas de todo tipo, cómputo de rendimientos, comparativas con grandes históricos). En muchas ocasiones, el usuario ha de compaginar la práctica asistencial con el manejo del ordenador. Usuarios que han de realizar procesos claves y que, en muchas ocasiones, no han recibido apenas formación sobre las aplicaciones que han de utilizar. Muchos puestos han de conectarse a periféricos sofisticados como grabadoras, escáners, cámaras, dispositivos de electromedicina, etc.

Parece claro que la parte servidora de nuestra aplicación será muy compleja, lo que no significa que la parte cliente vaya a serlo menos. Nos encontramos pues ante un entorno en el que las aplicaciones son difíciles de punta a punta, desconfiemos de aquellos que minusvaloren alguna de sus partes.

Éste es, a muy grandes rasgos, el ecosistema en el que nos movemos. Diseñar un sistema de información hospitalario es muy complicado. Conseguir que funcione no es únicamente un problema tecnológico. Como se ha repetido muchas veces, el factor organizacional puede ser aún más decisivo. Toda la organización en su conjunto, es decir, no sólo los informáticos y la dirección, ha de creer en las bondades del cambio que puede suponer la introducción de los sistemas de información en la gestión del hospital.

Un nuevo problema hay que añadir a esta ya de por si muy complicada situación. Un problema que, honestamente, creo que hay que poner encima de la mesa. Se trata de las relaciones entre médicos documentalistas e informáticos. No voy a entrar a describirlo en detalle, pues no es el lugar para ello. Soy informático, por lo que mi visión sería forzosamente parcial en este tema. Todas estas tecnologías son nuevas y es inevitable que algunas cosas cambien y que se necesitará un periodo de

adaptación. Pienso que documentalistas e informáticos estamos condenados a entendernos. Cuanto antes iniciemos un diálogo serio y lo más desapasionado posible sobre los problemas comunes, mucho mejor.

LA CONSTRUCCIÓN DE UN HIS

Antes de abordar la propuesta arquitectónica para la construcción de un HIS valdría la pena detenerse un momento en la justificación de tal empresa. ¿Para qué diseñar un HIS, por qué no adquirir uno? Es evidente que la adquisición de un HIS tiene muchas ventajas, su implantación siempre es difícil, su coste suele ser muy elevado, pero la gran mayoría de hospitales han optado por esta vía. "Outsourcing" es la palabra de moda. Sin embargo, esta forma de proceder también tiene inconvenientes que muy rara vez se mencionan. Pienso que el sistema de información de un hospital ha de devenir, en un futuro no muy lejano –si no lo es ya– en una pieza estratégica fundamental para el buen funcionamiento de la organización. El software que haya de implementar el núcleo funcional de la organización, no puede ser ni concebido, ni diseñado ni desarrollado completamente al margen de la organización. Va a ser demasiado decisivo en el futuro como para permitirnos el lujo de simplemente explicar los requerimientos funcionales que deseamos que tenga y retirarnos. La dirección operativa de la aplicación resultante también debe ser asumida por el hospital, o al menos éste habrá de participar activamente en ella. La organización también debe ejercer el control sobre cómo ha sido diseñado y desarrollado ese software, pues en caso contrario, la empresa puede tomar decisiones que pueden ser muy perjudiciales a medio y largo plazo para el hospital. Además, y esto se ha de decir de una vez por todas, programar bien no es nada fácil, y menos todavía la programación orientada a objetos a la que nos obliga Java. Existe de alguna forma la sensación de que la programación es casi lo de menos, algo trivial y mecánico, una tarea que puede ser llevada a cabo por personal sin gran cualificación técnica. En el mejor de los casos, esta presunción es, simplemente, fruto de la ignorancia. La calidad interna del software que se obtenga va a tener un fortísimo impacto en las posibilidades de crecimiento y evolución constante a la que se tendrá que enfrentar el sistema resultante. Muchas aplicaciones no funcionan sencillamente por lo mal desarrolladas y nulamente diseñadas que están. Por supuesto, no estoy diciendo que el hospital deba desarrollar solo el HIS, simplemente planteo la idea de que el hospital no puede estar al margen ni sobre la dirección, ni sobre el análisis, ni sobre el diseño ni sobre el desarrollo de este software tan importante. Entregarse completamente a una empresa puede devenir en un gran problema que se hace muy patente cuando las relaciones con la empresa se deterioran o si ésta desaparece. Que una empresa B deba asumir el mantenimiento de un software que

sólo conocía la empresa A es una tarea muy complicada, por muy bien documentada que la tengamos (seamos honestos, ¿tenemos actualizada la documentación del software?). Este cambio será más sencillo si la propia organización no es completamente ajena a él.

Supongamos, pues, que decidimos desarrollar un HIS. La primera y gran pregunta habrá de ser qué incluimos en él y que es lo que dejamos fuera. Existen demasiados condicionantes particulares para que exista una única respuesta. El hospital tendrá que ser realista sobre sus posibilidades, asumir que existen subsistemas en su organización muy especializados que tendrá que externalizar e integrar.

El núcleo de los procesos asistenciales, lo que ya se conoce como un clásico HIS (admisión, urgencia, hospitalización, quirófanos, citación, etc.) parece claro que habrán de estar dentro, también la estación médica y la de cuidados de enfermería (aunque existen buenas aplicaciones especializadas en estos aspectos), así como el grueso de la documentación que forma el grueso de la historia clínica (informes de alta, quirúrgicos, de seguimiento de consulta, de atención en Urgencias, etc.). Los sistemas departamentales ya es más difícil de analizar: diagnóstico por la imagen, laboratorios, anatomía patológica. Hay que evaluar el grado de acoplamiento entre estos subsistemas y el HIS central. La intercomunicación puede resultar insoportable conforme se vaya exigiendo cada vez más nivel de integración. En otros casos parece claro que la única opción es adquirir subsistemas de empresas externas: UCI, Reanimación, PACS, software de electromedicina.

Esbozaremos a continuación que componentes J2SE utilizaríamos para el desarrollo de nuestro HIS, más adelante analizaremos la problemática de integración con los subsistemas que hubieran quedado fuera. Finalmente veremos como ubicar el sistema resultante en un entorno aún más amplio.

Los puestos cliente

En primer lugar, es fundamental, que los clientes se presenten al sistema, mediante nombre de usuario y contraseña o, aún mejor, mediante una tarjeta que contenga la credencial certificada por alguna autoridad competente. La autentificación del usuario y el perfil de autorización deben ser resueltos desde el principio. Ello posibilitará que los puestos cliente estén convenientemente blindados y completamente controlados por el servicio de informática. Esto puede y debe ser realizado independientemente de si optamos por el sistema operativo Windows, Linux o Macintosh, teniendo la organización toda la política de usuarios centralizada, posiblemente en un LDAP o una base de datos relacional. La entrada del usuario al sistema se deberá validar contra este repositorio central, que será el que sepa que

priviliegios tiene el usuario y tenga constancia de sus entradas y salidas, algo a lo que estamos obligados por ley. Si dejamos la entrada libre y ponemos esta lógica de seguridad en las aplicaciones, siempre tendremos problemas con la desconfiguración de los puestos cliente. Esto no se ha hecho así desde el principio y puede costar un tiempo concienzar al colectivo de usuarios sobre la necesidad de acostumbrarse a la siempre engorrosa tarea de recordar contraseñas, tener que cambiarlas y, además, ver que determinadas cosas no se pueden hacer. Tenemos que ser serios: nos movemos, probablemente, en uno de los sistemas de información más confidenciales que existen. A cambio, deberíamos ser capaces de ofrecerles un sistema single-sign-on, esto es que los usuarios sólo tengan que autentificarse una vez, en lugar de estar obligados a presentarse una y otra vez a cada uno de los subsistemas en los que estuvieran autorizados.

El lector a estas alturas ya tendrá claro que abogo por construir verdaderas aplicaciones de escritorio para los clientes. Un hospital es una LAN, no es Internet, y la complejidad y seriedad de los puestos cliente merece que se le dé respuesta con las mejoras herramientas software. La interfaz de usuario es la que va a estar en contacto con los/as facultativos, enfermeros/as, administrativos/as y otros tipos de usuarios. Ellos son los verdaderos protagonistas de nuestra institución, ellos se merecen lo mejor.

Los argumentos en contra de aplicaciones de escritorio no son aplicables en nuestro entorno. Los puestos cliente tienen que estar bajo control de los servicios de informática. La posibilidad de actualización automática de la parte cliente de la aplicación mediante la invocación a través del protocolo JNLP y Java WebStart aniquila el último argumento serio que esgrimen los arquitectos pro-Web.

No entro aquí en la guerra SWT contra Swing, dos tecnologías rivales para la creación de aplicaciones Java en la parte del cliente. Las dos son buenas, sobre todo a partir de la versión 1.4 de J2SE en lo que a Swing se refiere. En cualquier caso, es muy positivo que una parte del equipo de desarrollo se especialice en este tipo de programación y que utilice frameworks específicos y patrones de diseño de forma consistente.

Los puestos cliente Java pueden tener, además, varios puertos abiertos, para que la parte servidora se ponga en contacto con ellos, por ejemplo para notificarle eventos que hubieran ocurrido en el sistema. El cliente, previamente, se habrá registrado como parte interesada en ser avisado sobre su ocurrencia. Imaginemos un monitor con la situación de una unidad de enfermería o de todo el hospital. El cliente escrito de esta manera puede estar permanentemente actualizado de forma desatendida, evitando refrescos innecesarios. En los entornos Web, este problema se solu-

cionaría con refrescos periódicos de la información, una solución no sólo menos elegante, sino que aumentará forzosamente el tráfico de la red.

Estos puertos cliente tienen otra utilidad importante, la de la administración y mantenimiento. El administrador de la aplicación puede obligar a un cierre de la aplicación cliente invocando un método de forma remota, el cliente tendrá prevista esta contingencia y habrá escrito el código necesario para salvar el estado de forma segura. Imaginemos, por ejemplo, a un facultativo que esté escribiendo un informe de alta y que, por labores de mantenimiento, necesitemos reiniciar la base de datos en la que se guarda dicho informe. Con este paradigma de programación, este problema puede ser resuelto limpiamente. ¿Por cierto alguien ha visto un procesador de textos Web? En un hospital se tienen que escribir muchos y muy sofisticados informes, si realmente deseamos que dicho informe esté generando automáticamente información estructurada para la historia clínica electrónica de forma transparente para el usuario.

La parte servidora

La parte servidora se muestra ante los clientes ofreciendo una serie de servicios, implementando un bean de sesión por cada caso de uso que se haya detectado en la fase de análisis. Posteriormente se pueden hacer mejoras y refinamientos a este patrón. Es una forma de empezar. Estos servicios ofrecen una API de alto nivel a los clientes que encapsulan los procesos y entidades nucleares del sistema.

Consideremos un servicio que se denomine Hospitalización, ofrecerá métodos invocables por cualquier aplicación que obtenga una referencia remota al bean de sesión que implemente la lógica relativa a los procesos asociados a la ocupación y liberación de camas, probablemente algunos de sus métodos se llamen: ingresar, trasladar, darDeAlta, reservar, etc. El cliente sólo deberá conocer los parámetros que ha de suministrar (entidades comunes en los clientes y parte servidora), los valores de retorno que le serán devueltos y capturar las excepciones que se puedan producir. Es una forma de implementar una arquitectura orientada a servicios (SOA) muy en auge actualmente. Veremos que nuestra propuesta pasa por diseñar SOA a dos niveles, a nivel de aplicación y de institución.

El modelo de dominio

Un sistema tan complejo como el nuestro merece un análisis pormenorizado del modelo de dominio. Se trata aquí de establecer la jerarquía de clases que han de representar las entidades de nuestro sistema, estableciendo no sólo sus atributos, su estado, sino también los métodos que estas entidades ofrecen al exterior, técnica-

mente su interface, algo que tenemos tendencia a olvidar. El modelo de dominio es el resultado del análisis orientado a objetos de nuestras entidades y de sus funcionalidades, no una simple enumeración de la estructura de su estado. Existen muchos modelos accesibles en la actualidad. Todos son muy complejos, pero es que la realidad también lo es.

Desde un punto de vista teórico, resulta muy interesante el modelo propuesto en www.openehr.org. La propuesta, ya no tan novedosa, consiste en establecer el modelado de las entidades mediante dos niveles de abstracción: el modelo de referencia y el de información. La idea es que el modelo de referencia representa, por explicarlo de alguna manera, los distintos tipos de ladrillos y las relaciones entre ellos con los que se nos estará permitido construir. El modelo de información utiliza estos ladrillos para la definición de entidades que representan ya la información de nuestro sistema, lo que se denomina arquetipos. Esta organización establece además un lenguaje, Archetype Definiton Language (ADL) para la definición formal de arquetipos a partir de otros arquetipos y de los constructos del modelo de referencia. Sin embargo, esta aproximación, tan sofisticada, aún nos parece demasiado complicada como para ser utilizada en sistemas reales que han de ser puestos en producción y ser mantenidos por una cultura y conocimientos aún muy alejados de esta forma de ver las cosas.

Otro modelo muy extendido es el RIM (Reference Information Model) de HL7. Si bien y como su nombre indica, su propósito es el de servir de modelo de referencia para HL7, también se ha utilizado como punto de partida para la definición de modelos de historia clínica electrónica.

La persistencia

Un debate muy de actualidad hoy en día es el de la forma de gestionar la persistencia en una arquitectura distribuida. Como vimos, J2EE ofrece la posibilidad de modelar las entidades de nuestro sistema mediante los beans entidad. Existen dos tipos de beans entidad: beans BMP (Bean managed persistence) y beans CMP (Container managed persistence). En los primeros, el programador ha de escribir el código de enlace entre las clases Java y la base de datos. En los segundos, se especifica este mapeo en un fichero XML, el descriptor de despliegue. Sin embargo, los beans entidad tienen algunos inconvenientes:

a) Un bean está pensado como un componente en si, se han de empaquetar las clases que lo conforman juntas, describir las opciones de despliegue y desplegarlo en el contenedor de EJBs, todo esto de forma autónoma los unos de los otros. Por esta misma razón, los programadores de beans de entidad no

lo tienen nada fácil para implementar la jerarquía de clases correspondientes al modelo de dominio.

b) Los beans entidad se pensaron como copias en caché de la información persistida en la base de datos. Ello se consigue gracias a que el contenedor está permanentemente actualizando el estado de los beans, independientemente de si esta información le interesa a un cliente o no, un mecanismo que requiere unos servidores desorbitadamente potentes para que no se vea mermado el rendimiento de forma más que apreciable.

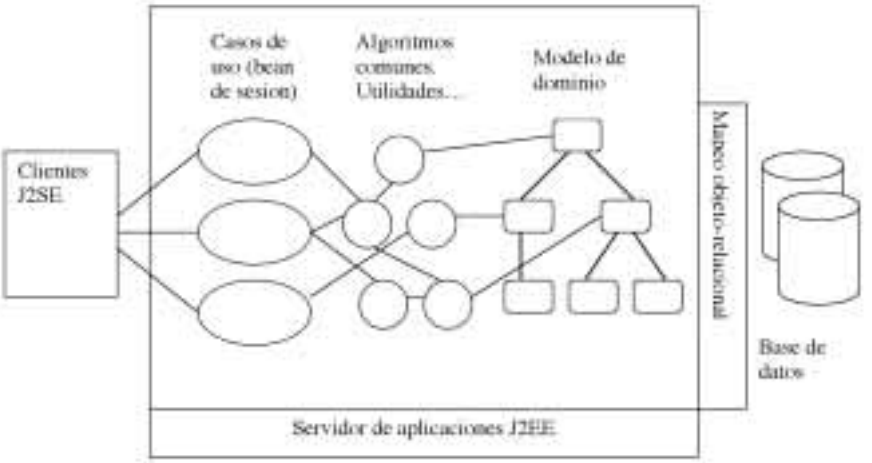
Esta situación ha llevado en muchas situaciones a abandonar la idea de crear un modelo de dominio, muchas aplicaciones escritas en Java son en realidad aplicaciones procedurales clásicas, la orientación a objetos es sólo aparente. La especificación EJB 3.0 promete solucionar estos problemas, pero ahora mismo es una promesa no es una realidad. ¿Aún suponiendo que EJB 3.0 sea la solución definitiva, cómo actuamos mientras tanto? Para solucionar esta grave problemática han aparecido distintas alternativas a los entity beans, herramientas de mapeo objeto-relacional compatibles con la existencia de un modelo de dominio jerarquizado: Hibernate, JDO o Castor.

Sin embargo, ninguna de ellas forma parte de J2EE. Aún así, en el estado en que se encuentra el debate y hasta la aparición de EJB 3.0 recomiendo el uso de alguna de ellas. JDO, Java Data Objects, es una especificación más del grupo JSR, es decir amparada por Sun. Los grandes papas del software y vendedores de servidores de aplicaciones J2EE (Oracle, Bea e IBM) han votado en contra de convertir la versión 2.0 de JDO en una especificación oficial de Java, pero no lo han conseguido. Parece que hay ciertos intereses comerciales en dicha decisión, un muy espinoso tema que no vamos a comentar aquí. Noticia muy reciente ha sido el anuncio de fusionar ambas tecnologías en una futura especificación, la JSR 220 que incorporará muchas de las ideas de JDO 2.0 para conseguir una EJB 3.0 muy reformada con respecto a la idea original.

Existen varias implementaciones de JDO (ver www.jdocentral.com) para más información sobre JDO, pero todas ellas son productos comerciales. Hibernate es un proyecto open source que tiene el mismo propósito. Su uso también se ha extendido bastante. Castor también es open source, pero no parece haber tenido el éxito de Hibernate.

Así, muy a grosso modo, la arquitectura que proponemos para la construcción de un posible HIS sería la siguiente:

Figura 4



Por supuesto existen una infinidad de detalles que no se reflejan en esta figura. Los mundos cliente y servidor, por ejemplo, no son disjuntos. La parte servidora se puede convertir de forma relativamente fácil en un modelo de sistema accesible por Web. Como vimos los beans de sesión sin estado se convierten casi automáticamente en servicios Web en un servidor J2EE 1.4. En cualquier caso, podemos colocar una capa Web por encima de nuestros casos de uso y posibilitar el acceso a nuestra parte servidora mediante el protocolo http.

No hay que ser integristas, en muchas situaciones, por ejemplo para la simple lectura de muchas filas de la base de datos, los clientes J2SE pueden acceder directamente a la base de datos, esquivando al servidor de aplicaciones. Los beans de sesión pueden evitar el paso por el modelo de dominio para la lectura de muchas filas, por ejemplo. El hecho de que el modelo de dominio esté formado por clases Java normales (llamadas POJOs en la literatura, Plain Ordinary Java Objects) va a facilitar enormemente el desarrollo.

Es importante tener estandarizado todo el desarrollo mediante el uso consistente de patrones de diseño de software. Un patrón de diseño es una solución genérica aplicable a distintos problemas particulares de software, pero que guardan una estructura común. Existe ya abundante literatura sobre el uso de patrones en aplicaciones J2EE y J2SE y su uso sistemático se considera un signo de calidad. Sin embargo, en algunas situaciones se ha llegado a abusar de los patro-

nes de software. Ninguna herramienta prefabricada puede sustituir a la inteligencia y el sentido común de un buen programador a la hora de resolver los problemas difíciles que frecuentemente se plantean en nuestros entornos. La estandarización de la construcción de la interface de usuario o de la forma de comunicarse con el servidor, la de intercambiar información entre los clientes y el servidor, el acceso al modelo de dominio, todas estas tareas son perfectamente "patronizables" (permítaseme el neologismo), pero ningún patrón de software va a evitar que alguien se tenga que calentar bien la cabeza para implementar algoritmos de citación en consulta externa que sean rápidos, fiables y diseñados con la suficiente granularidad para permitir ser reutilizados en otras partes de la aplicación (programación de quirófanos, por ejemplo). Las burbujas siguen sin compilarse solas.

INTEGRACIÓN DE SISTEMAS

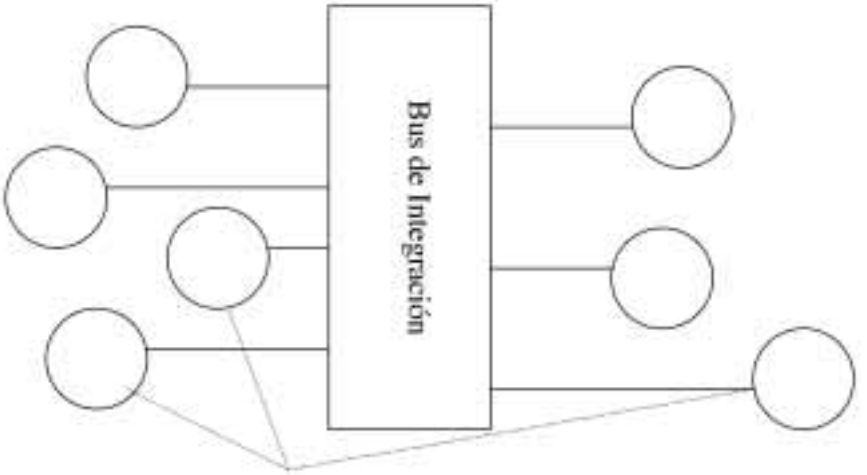
Suponganos, y es mucho suponer, que hemos conseguido desarrollar este HIS. Nos había quedado claro que una variedad de subsistemas quedarían fuera de nuestro HIS, serían productos de empresas externas, aplicaciones opacas en cierto sentido para la organización. ¿Cómo integramos todos estos subsistemas con el HIS central?

Por otra parte, cada vez se nos está exigiendo más integración entre todo el mundo no sanitario (gestión económica, personal, suministros, etc.) y la parte clínico-asistencial. Coste por proceso clínico, descentralización de las peticiones a suministros, contratos de gestión con los servicios, y muchos ejemplos más.

El problema de la integración de sistemas se ha de considerar como un proyecto en si, y dimensionarlo al alza de una vez. En muchas ocasiones se han ido resolviendo los problemas de integración uno a uno, sin una visión más general y a largo plazo. El resultado puede convertirse en un espagueti difícil de digerir.

Hace ya algún tiempo, que se vió la necesidad de concentrar de alguna manera toda la problemática de integración en un software central y específico. Lo que en los años 90 se denominó motor de integración, hoy se ha rebautizado con nombres como Integrador de Aplicaciones Empresarial (EAI). Cuando este software además de concentrador de mensajería, es capaz de ofrecer servicios a las aplicaciones dentro de un gran entorno, se viene denominando Bus de Servicios Empresarial (ESB). Así, el gráfico típico que aparece en las presentaciones de estos productos suele ser algo parecido a esto:

Figura 5



Aplicaciones heterogéneas

Como pieza estratégica fundamental, el software y hardware que dé soporte al Bus de integración habrá de compartir las mismas características de seguridad, fiabilidad, escalabilidad, robustez y rendimiento que las del servidor de aplicaciones J2EE que hubiéramos elegido en el caso del HIS.

El bus de integración no sólo tendrá que realizar tares de enrutamiento de mensajes, sino que tendrá también que incluir traductores o transformadores para los diferentes protocolos frecuentes en los entornos hospitalarios, como HL7, DICOM, EDI y otros, ofreciendo la posibilidad de encapsular este tipo de mensajes en invocaciones de Webservices, por ejemplo. Otra funcionalidad básica que debemos exigirle al bus de integración es la de encolamiento de mensajes, necesaria para poder garantizar la entrega de los mensajes, así como una fácil definición y gestión de colas, programación de alertas en caso de ciertos eventos. El servicio de informática debería ser el primero en enterarse de que algo falla, no el último.

También puede resultar un criterio muy interesante la posibilidad de interoperar a través del bus de integración con el correo electrónico o un fax. La posibilidad de transformar mensajes en los formatos PDF o RTF puede ser una característica importante para ello y que debemos evaluar.

Más recientemente, los ESB empiezan a ofrecer posibilidades más sofisticadas y prometedoras, como la monitorización de eventos a nivel de organización, o la coordinación u orquestación de servicios suministrados por más de una parte. Estas posibilidades pueden parecer irrelevantes, pero pueden ser muy interesantes, por ejemplo para coordinar una citación múltiple en laboratorio, diagnóstico por la imagen y una consulta externa, si son subsistemas diferentes, o para la definición de flujos de trabajo asociados a una guía clínica en la que intervendrán, seguramente, más de un subsistema. Resolver estos problemas, tan comunes por otra parte en un hospital, utilizando únicamente herramientas de mensajería puede ser muy complicado.

Otro problema que vemos en la práctica diaria en temas de integración es el de la definición de mensajes propios del centro, HL7 o DICOM tienen sus nichos propios. Aun siendo muy extensa su área de aplicación, no podemos esperar que nos proporcionen todos los mensajes que se pueden necesitar en un hospital, o para la conexión entre el hospital y el área de salud, otros hospitales o servicios centrales del gobierno autónomo, si estamos hablando de hospitales públicos o concertados. Estos mensajes se tendrán que hacer en base a un lenguaje propio, un ejercicio laborioso. La definición de este lenguaje se debería especificar al nivel más amplio posible dentro de la organización. Lo más razonable consistirá en definir un sublenguaje XML incrustándolo dentro de un mensaje SOAP. De esta manera podremos utilizar toda la infraestructura open source existente para el parseo y generación de servicios Web y nos evitaremos así complejos desarrollos y ofreceremos un estándar ya muy extendido, comprendido y aceptado para la comunicación con terceras empresas. Es muy conveniente definir este sublenguaje haciendo referencia a un modelo semántico de dominio, ausencia que siempre se ha echado de menos en las versiones 2.x de HL7 y que, en principio, se ha vista ya subsanada mediante el RIM. La idea aquí es asegurarse de que las dos partes involucradas en el intercambio de un mensaje puedan entender el contenido de éste suministrando toda la información de contexto necesario.

Últimamente se viene criticando el carácter de hub o concentrador del bus de integración, argumentando que se obtiene así un único punto de fallo para una tarea demasiado importante. Para ello, empiezan a proponerse arquitecturas en red de motores de integración. Sin embargo, parece que la configuración y el mantenimiento de este tipo de plataformas son aún demasiado complejos como para plantear una alternativa, necesitándose una fuerte dependencia de consultoría externa.

Además de escoger un servidor de alta disponibilidad para albergar el software necesario, se puede plantear el disponer de un bus de integración de backup, con la misma configuración que el principal. Los distintos subsistemas deberían ser capa-

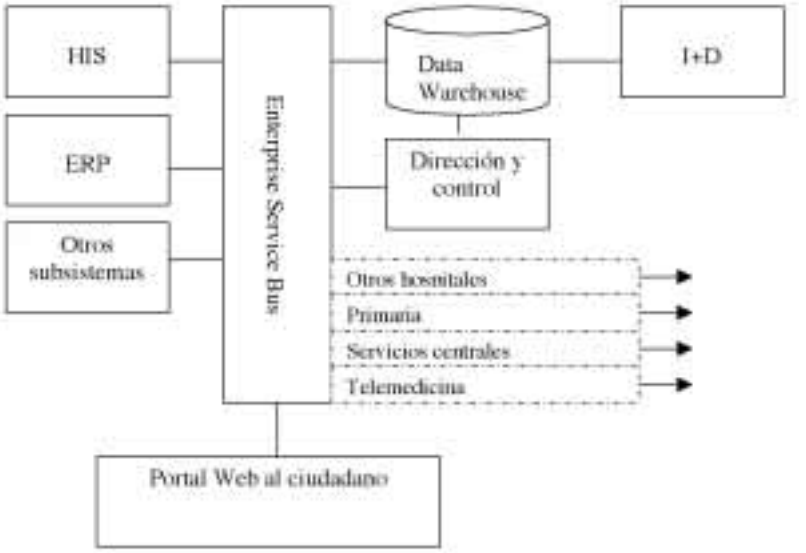
ces de reconfigurar la ruta al bus de integración en caliente. El bus de backup, por ejemplo, suministraría la nueva ruta a los distintos subsistemas, para que éstos, a partir de este momento, supieran donde enviar y recibir los mensajes.

Pienso que se ha de valorar mucho la experiencia del proveedor del bus de integración en el entorno sanitario. Muchos de ellos son demasiado genéricos y nuestro entorno es muy especial. Afortunadamente, ya vemos aparecer en el mercado verdaderos especialistas en el entorno de la sanidad, como Orion Rhapsody, Quovadx, OpenLink de Siemens o NEON de Sybase. Hay que considerar seriamente la posibilidad de elegir una de estas alternativas frente a los grandes monstruos como Weblogic Integration, MQ de IBM, Oracle, Microsoft BizTalk y otros.

PROPUESTA DE ARQUITECTURA DE UN SISTEMA DE INFORMACIÓN HOSPITALARIO

Falta unir todas las piezas que hemos venido comentando, añadir algunas más y dibujar un mapa que pudiera reflejar la arquitectura software de un hospital con un alto grado de informatización (el famoso hospital sin –muchos– papeles), un hospital que respondiera en gran medida a las necesidades que se nos vienen exigiendo desde hace tanto tiempo. La figura 6 refleja nuestra propuesta al respecto.

Figura 6



Hemos puesto el ERP y el HIS como sistemas independientes interconectados a través del bus de integración. Otras organizaciones pensarán que es mejor unificar estos dos (ya muy grandes en si) sistemas en un megasistema. Otros pensarán que el monstruo resultante pesa demasiado. En general, hay que ver cuales son las necesidades de comunicación entre dos sistemas, si esta necesidad es muy elevada, probablemente sea mucho más eficiente unificar los dos sistemas en uno. No hay que olvidar que la integración a través del bus es siempre un mal necesario, no una panacea que ha de resolver todos los problemas. En particular el ESB no es una herramienta que permita modularizar las aplicaciones. En cierta ocasión, escuchamos a un arquitecto de software plantear la posibilidad de diseñar aplicaciones distintas (con sus propias bases de datos) para los módulos de urgencias, admisión y consultas externas, unidas "sencillamente" por el omnipotente bus de integración. Esto es sencillamente un error.

Consideramos necesaria la adquisición de un Data Warehouse en el hospital. Este ha de servir de repositorio de datos consolidados y ser la fuente de información fundamental que dé soporte a los programas de investigación que se diseñen, descargando de este tráfico a los sistemas de producción. La dirección del hospital también hará un uso intensivo del Data Warehouse mediante herramientas de ayuda a la toma de decisiones, elaboración del cuadro de mandos integral, análisis de tendencias, data mining, etc. Sin embargo, la dirección también habrá de tener conexión on-line semidirecta a los sistemas de producción, si realmente desea estar al tanto de qué es lo que está ocurriendo ahora mismo en nuestro hospital (presión de urgencias, índice de ocupación, monitorización del presupuesto, etc.).

El omnipotente bus de integración también tiene la función de servir como puerta de entrada y salida a/desde el exterior. Por un lado para conectarnos con atención primaria, fundamentalmente para que atención primaria tenga la posibilidad de acceder a la parte de la historia clínica que pudiera ser de su interés. También la conexión entre hospitales sería una conexión inter-buses. No es en absoluto infrecuente que un mismo paciente tenga distribuida su historia clínica por varios hospitales, centros de especialidades y centros de salud. Una historia clínica electrónica puede y debe dar respuesta a esta situación tradicionalmente irresoluble.

Otro problema que ha aparecido desde hace ya cierto tiempo es la necesidad que tienen los servicios centrales de los gobiernos autónomos de obtener información de los hospitales y otras instituciones sanitarias para poder agregarla y analizarla. Lista de espera quirúrgica y de exploraciones centralizada, comparación de CMBDs y GRDs, estudios epidemiológicos, facturación intercentros, gestión de personal y muchos y muy complejos procesos se pueden incluir aquí. Desde el

intercambio de diskettes al envío de ficheros por correo electrónico (seguramente ilegal) se plantea desde hace tiempo basar esta comunicación a través de un intercambio de mensajes securizado entre los buses de integración de las distintas instituciones sanitarias y el que residiese en la propia Consejería de Sanidad.

Por otro lado, también las instituciones sanitarias necesitan que los servicios centrales les provean de información referente, por ejemplo, a tarjeta sanitaria y tipo de filiación de los pacientes, acceso a un repositorio centralizado de información común como centros, servicios, prestaciones, tasas, etc.

Finalmente, otro proyecto que considero esencial en un hospital como el que venimos describiendo es el portal al ciudadano. No se trata únicamente de diseñar la página web del hospital, sino de desarrollar una aplicación, web en este caso, por supuesto, que le ofrezca al ciudadano la posibilidad de interactuar con el hospital. Aún hay dudas razonables sobre el contenido funcional de esta aplicación, pero lo que parece evidente es que la tendencia deberá ir por el camino de dotarla de más capacidad de producir eventos en nuestros sistemas, no sólo de servir de un mero panel informativo. La posibilidad de que los usuarios dispongan de certificados digitales ayudará en gran medida a ello. Muchos gobiernos autónomos han reconocido estas posibilidades y se han convertido ya en Autoridades Certificadoras oficiales.

RESUMEN Y CONCLUSIONES

Para la construcción de una gran aplicación distribuida y utilizando las últimas tecnologías software, J2EE ofrece una especificación lo suficientemente versátil para afrontar la gran complejidad que plantea un sistema de información hospitalario.

Un servidor de aplicaciones J2EE es una implementación de esta especificación, un producto altamente sofisticado que requerirá de un soporte y mantenimiento especializado. El servidor deberá ofrecer un contenedor Web para posibilitar el acceso a nuestra aplicación mediante un cliente http, y un contenedor de EJBs para modelar los procesos y entidades que definen nuestro sistema.

Demasiado a menudo se confunde una aplicación J2EE con una aplicación web, cuando la realidad es que la especificación ofrece toda una serie de paradigmas arquitectónicos que incluyen otras alternativas. Una de ellas, perfectamente aplicable a un entorno hospitalario consiste en la utilización de clientes J2SE conectados a la parte servidora a través de IIOP. La parte servidora expondría los casos de uso a los clientes mediante Enterprise Java Beans de tipo sesión. Para la implementación de los servicios ofertados, los EJBs utilizan algoritmos comunes y otras clases auxiliares. Toda la aplicación, además, hará uso de un modelo de dominio, la jerar-

quía de clases que representan las entidades de nuestro sistema. Sin embargo, los EJBs de tipo entidad no han resultado ser una tecnología útil y eficaz para la gestión de la persistencia de todo un modelo de dominio complejo en una base de datos relacional. El autor recomienda que, hasta que la especificación EJB 3.0 sea una realidad, se utilice alguna buena implementación de JDO (Java Data Objects) para conseguir la deseada persistencia del modelo de dominio.

Entre el dilema de construir un sistema de información hospitalario o adquirir un producto comercial, abogamos por la primera opción. No creemos en la existencia de un hospital estándar y la parametrización y adaptación de un producto externo a un entorno concreto siempre será costosa, en todos los sentidos del término. Además, el producto final obtenido siempre se verá condicionado por el hecho de no haber nacido desde la organización. Por supuesto, un hospital no va a poder desarrollar solo un sistema de información hospitalario integral. Por esta razón, necesitaremos de empresas de software que nos ayuden a ello. Sin embargo, pensamos que el modelo de externalización del desarrollo que se ha seguido normalmente no es el correcto. El sistema de información de una organización hospitalaria ha de devenir en una pieza demasiado estratégica como para limitarnos a dirigir y coordinar a agentes externos. La organización ha de contar con técnicos que no vean las aplicaciones como meras cajas negras. El diseño e incluso la codificación de las aplicaciones que se desarrollen ha de ser al menos conocido y codecidido por técnicos informáticos del hospital y garantizar así un producto con la calidad y bondades suficientes como para responder a las necesidades actuales, pero también las que se requerirán de ellas a largo plazo.

Obviamente, no todos los módulos de un gran sistema de información hospitalario van a poder desarrollarse. Algunos de ellos, PACs, sistemas de electromedicina, laboratorios, etc., deberán adquirirse fuera. Por este motivo, resultará fundamental para el éxito de una empresa de tal envergadura el disponer de un motor de integración que nos permita a medio plazo la construcción de un bus de servicios hospitalario. Este bus servirá tanto para la integración dentro del hospital como para su conexión con el exterior: atención primaria, servicios centrales, otros hospitales y centros.

GLOSARIO DE SIGLAS

ADL: Archetype Definition Language. Es un lenguaje formal para la definición de arquetipos.

BMP: Bean Managed Persistence. Se utiliza cuando es el programador el que escribe el código para persistir el estado de los beans entidad.

CMP:Container Managed Persistence. El contenedor persiste el estado de los beans entidad en base a lo especificado en los descriptores de despiegue.

EJB: Enterprise JavaBeans. Son una colección de clases e interfaces Java que se ejecutan dentro de un contenedor específico para ellos y con los que podemos representar procesos y entidades accesibles de forma remota.

ESB: Enterpise Service Bus. Pieza software que posibilita que las aplicaciones pueden publicar y utilizar servicios de otras aplicaciones.

IIOP: Internet Inter Object Request Broker Protocol. Es un protocolo para la comunicación entre objetos distribuidos.

J2EE: Java 2 Enterprise Edition. Es la especificación para el desarrollo de grandes aplicaciones distribuidas en varias capas.

J2SE: Java 2 Standard Edition. Especifica la plataforma de ejecución de programas Java.

JCA: Java Connector Architecture. Es una de las especificaciones hijas de J2EE. En ella se indica como un proveedor de un sistema no J2EE puede interactuar con un servidor de aplicaciones J2EE.

JDBC: Java DataBase Connectivity. Es un API genérica mediante la que un programador Java puede comunicarse directamente con una base de datos relacional.

JDO: Java Data Objects. Esta especificación no está incluida en J2EE, pero es la tecnología más prometedora para poder mapear un modelo de dominio a una base de datos relacional o de otro tipo.

JMS: Java Message Service. Es otra API J2EE que sirve para interactuar con un proveedor de servicios de mensajería.

JNLP: Java Network Launching Protocol. Mediante la implementación de este protocolo ofrecida mediante Java WebStart, clientes J2SE pueden actualizarse automáticamente a través de un repositorio centralizado accesible por http.

JRMP: Java Remote Method Protocol. Es el protocolo nativo Java para la interoperación entre instancias de clases Java que se ejecutan en diferentes máquinas virtuales.

JSP: Java Server Pages. Código html con referencias explícitas e implícitas a código Java y que al ser desplegadas dentro del contenedor Web se convierten en servlets.

MVC: Model View Controller. Es el patrón de diseño software más conocido. Se trata de particionar un sistema en tres partes: su vista, su estado o modelo y el controlador que hace de puente entre ellas.

.NET: Es la alternativa Microsoft a J2EE. Limitada a entornos Microsoft.

POJO: Plain Ordinary Java Object. Representa una instancia normal de una clase Java, en contraposición a otras construcciones más sofisticadas, como los Enterprise JavaBeans, por ejemplo.

Servlet: Es una clase Java que al ser desplegada en un contenedor Web sirve de puente entre el mundo http y el mundo Java.

SOA: Service Oriented Architecture. Es la arquitectura software en la que los sistemas ofrecen servicios a otros sistemas a través de un bus y poder, así, construir otros sistemas más elaborados.

SOAP: Simple Object Access Protocol. Es un lenguaje XML que sirve de protocolo en la tecnología de servicios Web.

Swing: No es ningún acrónimo. Son todas las clases que ofrece J2SE para la construcción de sofisticadas interfaces de usuario.

SWT: Simple Widget Toolkit. Hubo una época en que Swing tuvo fama de lento y complicado. IBM sacó esta alternativa para la construcción de interfaces de usuario basada también en Java.

BIBLIOGRAFÍA Y ENLACES RELACIONADOS

1 La especificación J2EE 1.4 se encuentra en http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf

2 Weaver J. L., et al. Beginning J2EE 1.4: From Novice to Professional. Apress; 2004.

3 Roman Ed., et al. Mastering Enterprise JavaBeans. Wiley; 2001.

4 Marinescu F. EJB Design Patterns. Wiley; 2002.

5 Johnson R. Expert One-on-One J2EE Designs and Development (Programmer to Programmer). Wrox; 2002.

6 Alur D., Crupi J., Malks D. Core J2EE Patterns. Best Practices and Design Strategies. Prentice Hall; 2003.

7 Chappell D. Enterprise Service Bus. O'Reilly; 2004.

8 Hohpe G., et al. Enterprise Integration Patterns. Designing, Building and Deploying Messaging Solutions. Addison-Wesley Professional; 2003.

9 Fowler M., et al. Patterns of Enterprise Application Architecture. Addison Wesley; 2002.

10 McGovern J., et al. Java Web Services Architecture. Morgan Kaufmann; 2004.

11 Chappell D., Jewell T. Java Web Services. O'Reilly; 2002.

12 Jordan D., Russell C. Java Data Objects. O'Reilly; 2003.

13 Robinson M., Vorobiev P. Swing. Manning; 2003.

14 Amatayakul M. Electronic Health Records. A practical Guide for Professionals and Organizations. American Health Information Management Association; 2004.

15 Walker J. M., Bieber E. J., Richards F. Implementing an Electronic Health Record System. Springer Verlag; 2004.

16 www.openehr.org

17 www.hl7.org

18 www.theserverside.com

19 www.jdocentral.com

20 community.java.net/javadesktop

21 www.jguru.com