Charla 2: SOA e integración

Índice

Resulta habitual que con el paso del tiempo las organizaciones tengan implementados sus sistemas de información "repartidos" entre diferentes departamentos, con diferentes arquitecturas, tecnologías y diferentes plataformas software y/o hardware. Esta heterogeneidad, que por desgracia es inevitable, ocasiona unos elevadísimos costes de mantenimiento, ya que los negocios modernos necesitan adaptarse continuamente a las exigencias de los clientes y el mercado, y lo tienen que hacer de una forma eficiente, para poder seguir siendo competitivos y mantener sus negocios.

La integración de aplicaciones pretende conseguir un modelo de arquitectura software que sea capaz de adaptarse a un modelo de negocio siempre cambiante. A la hora de poner en práctica la integración de aplicaciones, las arquitecturas software tradicionales han demostrado ser claramente poco adecuadas para implementar soluciones efectivas al problema de integración. En este sentido, las arquitecturas orientadas a servicios (SOA: *Service Oriented Architecture*) proporcionan una solución relativamente barata y efectiva a dicho problema, por tratarse de una arquitectura flexible, extensible, y que encaja bien con las aplicaciones *legacy* existentes.

En esta charla, por lo tanto, hablaremos de las arquitecturas SOA en un contexto de integración de aplicaciones, por lo que en primer lugar explicaremos la problemática de la integración de aplicaciones, para entender cómo SOA puede abordar los problemas que ésta plantea.

1. Integración de aplicaciones

Las empresas necesitan, cada vez más, un fácil acceso a la información por parte de sus aplicaciones. Esta necesidad presenta nuevos retos para el desarrollo de aplicaciones. La facilidad de acceso a los datos es poco probable que se consiga mediante aplicaciones separadas (*stand-alone*) usadas por la mayoría de las empresas, debido a la dificultad que representa el compartir datos entre ellas. Sin embargo, muchas empresas no pueden permitirse el lujo de retirar o reemplazar sus aplicaciones *stand-alone* de un día para otro debido a que juegan un papel crítico en el funcionamiento de la empresa, y a menudo no es rentable desarrollar de nuevo la aplicación en su totalidad. Recordemos que una aplicación *stand-alone* es aquella que se puede ejecutar sin necesidad de ningún elemento soporte (por ejemplo un navegador).

Por otro lado, a medida que pasa el tiempo, las empresas necesitan introducir nuevas aplicaciones y sistemas. Estas nuevas soluciones se basan normalmente en arquitecturas modernas, que difieren de forma significativa de las arquitecturas usadas por aplicaciones que ya tienen años de uso (aplicaciones *legacy*). A menudo las aplicaciones modernas se adquieren en forma de componentes, que se integran en una aplicación más grande. Estas nuevas aplicaciones necesitan integrarse en el sistema existente para que la información que contienen esté disponible y sea accesible.

En estas situaciones, la integración de aplicaciones corporativas (EAI) adquiere una gran importancia, permitiendo que una empresa integre sus aplicaciones y sistemas existentes

y sea capaz de añadir nuevas tecnologías y aplicaciones al nuevo conjunto.

La integración de aplicaciones no es una tarea fácil; de hecho se ha convertido en uno de los problemas más difíciles con los que se enfrenta el desarrollo de aplicaciones para la empresa desde hace pocos años. Los mayores retos se presentan en la integración de diferentes dominios, arquitecturas y tecnologías. Además, los requerimientos de los sistemas de información van creciendo significativamente y cambiando continuamente, por ello los proyectos de integración deben realizarse en el menor tiempo posible, entregar sus resultados rápidamente, y adaptarse a estos requerimientos siempre cambiantes.

1.1. ¿Qué es la EAI?

La integración de aplicaciones corporativas (*Enterprise Application Integration*, EAI) es básicamente un nuevo nombre para el proceso de integración en el que las empresas han estado trabajando durante años. EAI hace referencia a una integración global y sistemática.

Antes de definir de forma más precisa la EAI, consideraremos con más detalle cómo surge la necesidad de integrar las aplicaciones de una empresa.

En el pasado, las aplicaciones se ideaban como soluciones individuales a problemas aislados. Los arquitectos software no pensaban en esas aplicaciones como partes de un sistema de información más amplio de la empresa. Esta es la razón de por qué la mayoría de aplicaciones antiguas permiten una interoperabilidad muy limitada con otras aplicaciones. Para modificar dichas aplicaciones y hacerlas más interoperables, se necesita un buen conocimiento del desarrollo de la aplicación y de los procesos lógicos que implementa. Incluso hoy en día, algunas aplicaciones se desarrollan sin tener en cuenta (o muy poco) cómo conectarlas con otros sistemas. Por estas razones, EAI es relevante tanto para las aplicaciones ya existentes, como para aplicaciones modernas.

La importancia de la EAI radica en las expectativas del negocio. Desde el punto de vista del negocio, el objetivo es maximizar los beneficios de cada aplicación y el sistema de información como un todo. Las aplicaciones separadas no pueden cumplir este requerimiento.

Parte del problema es que los datos se encuentran particionados y replicados entre las diferentes aplicaciones. Cada aplicación modela los datos de forma diferente, de acuerdo con las necesidades de la aplicación, no de la empresa. Esto hace que sea difícil "ensamblar" los datos de diferentes aplicaciones, ya que probablemente usarán diferentes tecnologías, aplicaciones y bases de datos para acceder a ellos. En la Figura 1 se muestra un dominio típico de empresa. En este escenario, podemos encontrar "islas" de funciones y datos, y cada una de ellas existe con su dominio del problema de forma separada.

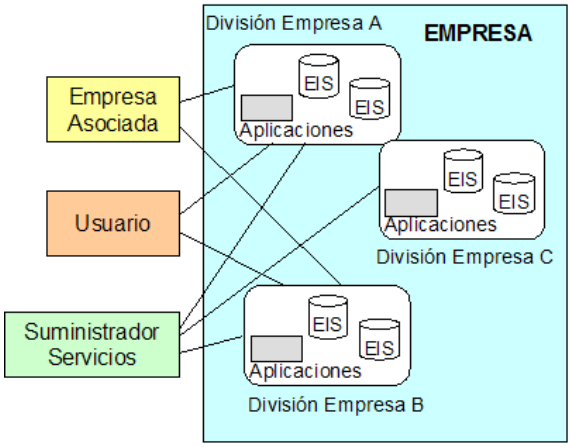

Figura 1. Dominio de negocio típico.

Las empresas necesitan modernizar y mejorar la funcionalidad de sus sistemas de información para seguir siendo competitivas (los gestores ven el sistema de información como una herramienta que pueden utilizar para maximizar los beneficios de la empresa).

Mejorar la funcionalidad de un sistema de información puede hacerse de varias maneras: (a) La forma más obvia consiste en reemplazar las aplicaciones antiguas por una solución más reciente. Incluso aunque a primera vista parezca una solución atractiva, es inaplicable en la mayoría de los casos. Reemplazar sistemas existentes con nuevas soluciones siempre requiere más tiempo y dinero de lo que en principio estaba planificado, incluso teniendo en cuenta los escenarios más pesimistas. Incorporar todas las peculiaridades y casos especiales en el software requiere un conocimiento del sistema existente. Este conocimiento es mucho más fácil de adquirir si la empresa tiene sus procesos del negocio bien documentados, cosa que no suele ocurrir en la práctica. Además, a menudo, no hay una única persona que conozca el proceso en su totalidad; (b) Otra alternativa es la de introducir soluciones comerciales. Esta solución probablemente no sea factible, ya que cada empresa tiene una forma particular de realizar los negocios, y que las distingue. El adaptar soluciones comerciales a la empresa requiere tiempo, y tiempo significa dinero. Introducir nuevos sistemas requiere también entrenar y educar al personal.

Una alternativa factible para mejorar la funcionalidad del sistema de información consiste en conseguir una forma estándar para reusar sistemas existentes pero integrándolos en un sistema global de información de la empresa más amplio. En este sentido, EAI proporciona una metodología estándar para la comunicación entre aplicaciones y fuentes de datos.

Ahora daremos una definición más precisa del término EAI. Realmente, la **definición de EAI** varía dependiendo del punto de vista:

Desde el **punto de vista del negocio**, EAI representa la ventaja competitiva que consigue una empresa cuando todas las aplicaciones están integradas en un sistema de información único, capaz de compartir información y soportar flujos de trabajo del negocio (*business*

workflow). La información debe compartirse a menudo desde diferentes dominios y ser integrada en un proceso de negocio. Sin EAI, si bien la información requerida puede que exista y esté disponible en algún lugar y de alguna forma en una aplicación, para los usuarios típicos es prácticamente imposible tener acceso a ella *on-line*.

Desde el **punto de vista técnico**, EAI se refiere al proceso de integrar las diferentes aplicaciones y los datos sin tener que modificar demasiado las aplicaciones existentes. EAI debe realizarse utilizando métodos y actividades que permitan que dicho proceso sea efectivo en términos de coste y tiempo. Aquí es donde veremos que SOA puede ayudar.

Existen muchos ejemplos reales de EAI, particularmente en la industria de la banca y servicios financieros, así como en telecomunicaciones. Por ejemplo, AT&T comenzó como un suministrador de servicios de telefonía, a los que añadió los servicios de televisión por cable y otros servicios inhalámbricos. Más tarde se convirtió en un suministrador de banda ancha. La compañía ha crecido absorbiendo a otras compañías y adquiriendo otros negocios. Como resultado de su crecimiento, y antes de sus planes actuales de dividirse en cuatro compañías diferentes, AT&T necesitó integrar sus servicios *on-line* a sus clientes. Tuvo que integrar sus presentaciones de facturas a todos los servicios, los pagos de los servicios, y la totalidad de los servicios a los clientes. Esto supuso una integración del acceso a las aplicaciones existentes que proporcionaban dichos servicios.

En el proceso de integración de procesos del negocio y los datos, EAI abarca tanto la distribución de dichos procesos y datos, como el concepto de reutilización de módulos. Y lo que es más importante, EAI aborda esta integración como un proceso separado de las diferentes aplicaciones. Es decir, alguien puede integrar varias aplicaciones, incluyendo las fuentes de datos subyacentes, sin tener que comprender o conocer los detalles propios de cada aplicación.

No todas las empresas se benefician de igual forma del proceso de integración. Por supuesto, cuando una empresa tiene únicamente un número limitado de sistemas, que comparten tecnología y arquitectura, la integración es mucho más fácil. A la inversa, puede haber sistemas formados por grandes aplicaciones diferentes, distribuidas geográficamente en varias plataformas, algunas de las cuales tienen funcionalidades comunes. En este caso, la empresa se beneficiará considerablemente del proceso de integración.

Además, para realizar una integración de forma exitosa es necesario evaluar el estado de las aplicaciones existentes, analizar los requerimientos del sistema, y comparar la situación actual con las metas fijadas y los requerimientos actuales. Este estudio debe realizarse caso por caso, ya que no es fácil definir procedimientos universales.

Puesto que hemos visto que la comprensión de los sistemas existentes es vital para una integración efectiva, repasaremos el concepto de sistemas de información, y analizaremos más la arquitectura de dichos sistemas.

1.2. Sistemas de Información de Empresas (EIS)

Antes de entrar en detalles sobre la EAI, es útil entender la definición de un sistema de información de empresa (**EIS**: *Enterprise Information System*). Una empresa requiere ciertos procesos de negocio y datos subyacentes para ejecutar dichos procesos. Un EIS abarca los procesos del negocio y la infraestructura para las tecnologías de información (**IT**: *Information Technology*). Los procesos del negocio incluyen aplicaciones para la gestión del procesamiento de nóminas, gestión de inventario, control de producción de fabricación, y la contabilidad financiera.

Definimos un sistema de información de empresa como un sistema con una o varias aplicaciones que proporciona la estructura de información para la empresa. Un EIS proporciona un conjunto de servicios a sus usuarios, que pueden estar disponibles en distintos niveles de abstracción (por ejemplo a nivel de sistema, nivel de datos, o nivel de negocio).

En la Figura 2 se puede apreciar gráficamente esta situación. En el entorno mostrado, las aplicaciones residen en un servidor de aplicaciones. Dicho servidor tiene una infraestructura particular dependiente del vendedor, contemplando en particular los servicios de procesamiento de transacciones, seguridad y balanceo de carga. Las aplicaciones incluidas en el servidor pueden ser suministradas por diferentes vendedores, o pueden desarrollarse en el departamento IT de la empresa. Las aplicaciones están escritas en varios lenguajes, como COBOL, C, y C++. Los clientes acceden a las diferentes aplicaciones mediante interfaces (**API**s: *Application Programming Interfaces*). Un API es alguna rutina que permite a un cliente realizar operaciones tales como crear una orden de compra o actualizar un registro de usuario. La interfaz de acceso a los datos permite acceder al almacenamiento de los datos (ficheros o base de datos relacional). Los interfaces de objetos del negocio son abstracciones que representan la lógica específica del negocio para acceder a las funciones y los datos.

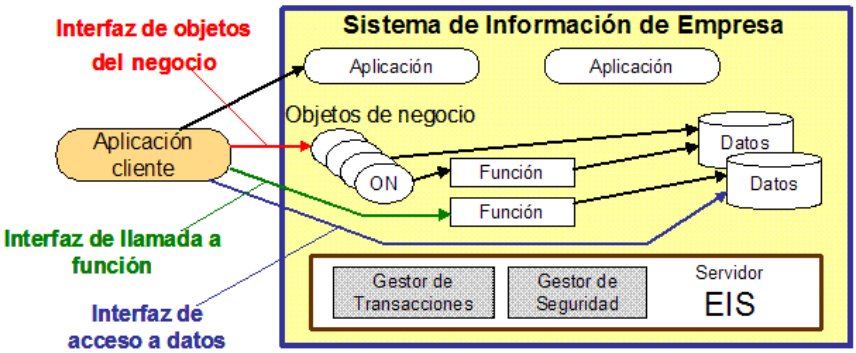

Figura 2. Entorno Sistema de Información de Empresa.

Hay muchas y diferentes aplicaciones que pueden catalogarse como un EIS. Por ello los EIS varían unos de otros, incluso dentro de la misma empresa. Una empresa puede

desarrollar varios EISs a lo largo del tiempo, dependiendo de las necesidades particulares del momento. Por ejemplo, una empresa puede comenzar con un sistema de fabricación. A medida que pasan los años y la empresa crece, se añaden otros paquetes como contabilidad, atención al cliente y recursos humanos. Algunos servicios puede incluirlos en la plataforma en donde residen las operaciones relacionadas con la fabricación, sin embargo, otros requerirán ser desarrollados con una plataforma o arquitecturas diferentes. La empresa no solamente añade nuevos sistemas software, sino que adquiere *hardware* adicional que puede ser completamente diferente de su configuración original. Es fácil ver que cuando una empresa opera durante largo tiempo, puede estar usando EIS que han sido desarrolladas e instaladas sobre diferentes plataformas y arquitecturas.

1.3. Arquitectura de aplicaciones existentes

Los proyectos EAI tienen que tratar con aplicaciones (sistemas de información) existentes. Algunas de las aplicaciones existentes son antiguas, basadas en tecnologías no consideradas actualmente como "estado del arte", con lenguajes de programación en desuso, y sobre plataformas anteriores a las actuales, estas aplicaciones se conocen con el nombre de aplicaciones o sistemas ***legacy***. Podemos considerar dos tipos de sistemas *legacy*: los sistemas monolíticos tradicionales (basados principalmente en sistemas *mainframe*), y los sistemas cliente/servidor (basados mayormente en PC's).

Una **arquitectura monolítica** contiene la presentación, la lógica del negocio y los datos en la misma aplicación. Normalmente se trata de aplicaciones con un diseño estructurado e implementadas con lenguajes estructurados como COBOL, Fortran y PL/1 entre otros. En la mayoría de casos, los datos se almacenan en ficheros usando diferentes formatos binarios. Los sistemas contienen miles de líneas de código que han sido desarrolladas y mantenidas a lo largo de los años por diferentes desarrolladores. Usualmente no se dispone de documentación, y en el peor de los casos, tampoco del código fuente. La mayoría de dichos sistemas residen en ordenadores *mainframe*. Huelga decir que inicialmente no había ningún tipo de integración entre las diferentes aplicaciones monolíticas. Sin embargo, las soluciones de integración sobre dichos sistemas se comenzó a realizar hace unas tres décadas.

Un primer intento de integración consiste en enlazar las diferentes aplicaciones con una base de datos centralizada (ver Figura 3). Dicha base de datos normalmente no almacena todos los datos de todas las aplicaciones (debido a restricciones de capacidad y rendimiento), por lo tanto éstas todavía necesitan de bases de datos locales. Este tipo de integración requiere una re-programación de cada aplicación.

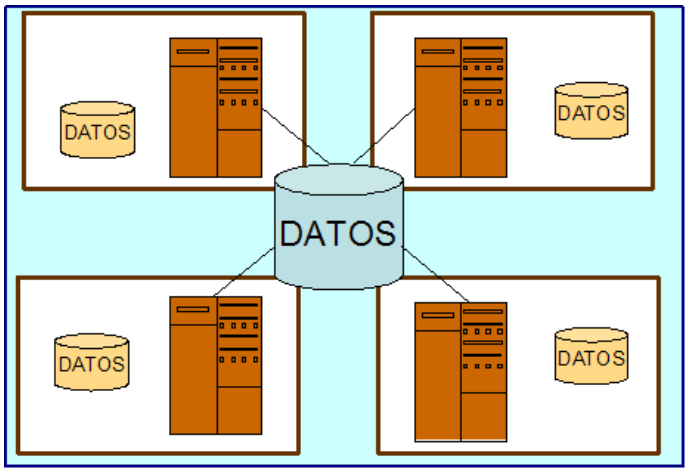

Figura 3. Integración con BD centralizada.

Desde una perspectiva técnica, esta integración no es demasiado difícil cuando solamente hay una plataforma en la empresa, y no hay demasiadas aplicaciones a integrar. Desafortunadamente, hay un gran número de empresas que no pertenecen a esta categoría.

Posteriormente, en la década de los 80, el proceso de reemplazamiento de los *mainframes* por PCs lleva consigo algunos grandes cambios en las arquitecturas de las aplicaciones, pero complica todavía más las cosas. No se dispone de estrategias para estandarizar las plataformas y tecnologías a usar, con lo que cada departamento es libre de realizar su elección, usualmente basada en preferencias personales, y no en los objetivos de la empresa en su totalidad. Por un lado las necesidades de integración se hicieron cada vez más crecientes, por otro lado, los PCs interconectados comenzaron a sustituir a los terminales "mudos". A medida que fue necesario compartir los datos entre los usuarios, las arquitecturas monolíticas se volvieron inviables.

La solución a estos problemas fue la arquitectura **cliente/servidor** que separa las aplicaciones en dos capas (ver Figura 4). En la mayoría de los casos, la gestión de los datos se separó del resto de la aplicación (ver Figura 4a). Esto significaba que la lógica del negocio y la interfaz del usuario todavía permanecía unida, y en demasiados casos, interconectadas. Debido a que el cliente realizaba la mayor parte del procesamiento implicado en la lógica del negocio, a este tipo de aplicaciones cliente/servidor se les ha denominado **clientes ricos** (*fat clients*). Los datos usualmente residían en uno o varios servidores dedicados de gestión de base de datos (DBMS). Otra posibilidad para los sistemas cliente/servidor fue separar la lógica del cliente y situarla en el servidor (ver Figura 4b). En este escenario, el código en la parte del cliente es mucho más sencillo, y a dichos clientes se les ha llamado **clientes ligeros** (*thin clients*). La lógica del negocio se almacenaba en el servidor de la base de datos (en forma de procedimientos con sentencias SQL), o en un servidor fuertemente acoplado a la base de datos. La mayoría de los gestores de bases de datos soportan procedimientos almacenados, si bien hay variaciones en la sintaxis y en sus capacidades, creando así otro problema potencial: el traslado de

esta lógica de un DBMS a otro es una tarea extremadamente ardua, si no prácticamente imposible.

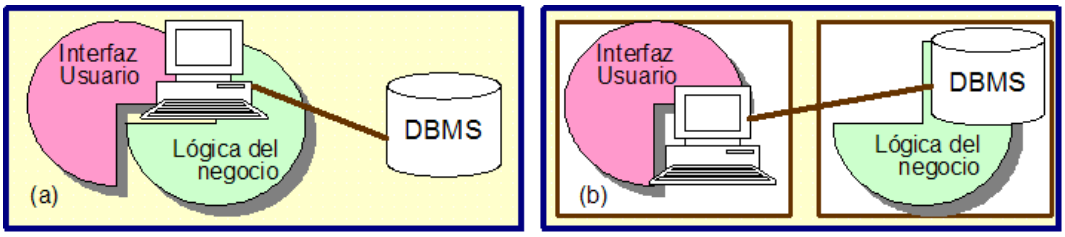

Figura 4. Arquitectura cliente/servidor. (a) Cliente rico. (b) Cliente ligero

La mayor parte de las nuevas aplicaciones, a partir de entonces se desarrollaron siguiendo la arquitectura cliente/servidor, pero las soluciones monolíticas permanecieron, haciendo así crecer la necesidad de integración. Además, la arquitectura cliente/servidor implica un estrecho acoplamiento entre la aplicación y la base de datos.

El número de aplicaciones diferentes en una misma empresa fue creciendo. Si bien diferentes aplicaciones resolvían problemas diferentes, en muchos casos había algún solape funcional entre ellas, con lo que se podía dar el caso de datos similares residentes en bases de datos diferentes, con nombres diferentes. Para una integración efectiva, el acceso a los mismos datos no resuelve todos los problemas. Dicho acceso a los datos y su posible transferencia entre bases de datos no es suficiente debido a la dificultad que entraña el análisis de dichos datos para obtener información del negocio valiosa para el soporte de decisiones. Tampoco es posible reusar todas las funcionalidades de otras aplicaciones. Sin la reusabilidad de las funciones la única posibilidad consiste en implementar las funcionalidades en diferentes aplicaciones, incrementando la complejidad de nuevo (especialmente a la hora de mantener las aplicaciones). Duplicar la funcionalidad es necesario debido a que si accedemos a los datos en la base de datos directamente, evitamos implementar reglas del negocio en la aplicación original; pero esto significa que tenemos que ser extremadamente cautos debido a que un acceso directo a la base de datos puede amenazar la integridad del sistema.

Para resolver el problema del análisis de grandes cantidades de datos almacenados en aplicaciones diferentes, y permitir el compartir funcionalidades de las aplicaciones, se utilizan los *datawarehouse* y los *sistemas ERP*.

A principios de los 90, la integración es una cuestión relevante desde dos puntos de vista distintos. Por un lado tenemos el uso de *data warehouses*. Un ***data warehouse*** es un repositorio central para todos los datos significativos que las aplicaciones de una compañía obtienen. Debido a la variedad de datos bien estructurados contenidos en ellos, los *data warehouses* fueron (y todavía siguen siendo) efectivos como sistemas de soporte de decisiones *off-line*, y útiles siempre que puedan obtenerse datos "en bruto" y analizarlos para derivar conclusiones a partir de ellos.

Los datos de las bases de datos de las aplicaciones se envían al *data warehouse* a intervalos de tiempo predeterminados, normalmente una vez por día. Esto hace que los

data *warehouses* no sean la elección perfecta para la integración, especialmente cuando no podemos permitirnos el trabajar con datos que pueden estar anticuados. Los procedimientos asociados con un sistema *data warehouse* se asemejan a los de la Figura 5, en donde los datos desde diferentes bases de datos son extraídos, transformados y cargados en un data *warehouse* central. Las reglas para estas tres fases se definen como meta-datos.

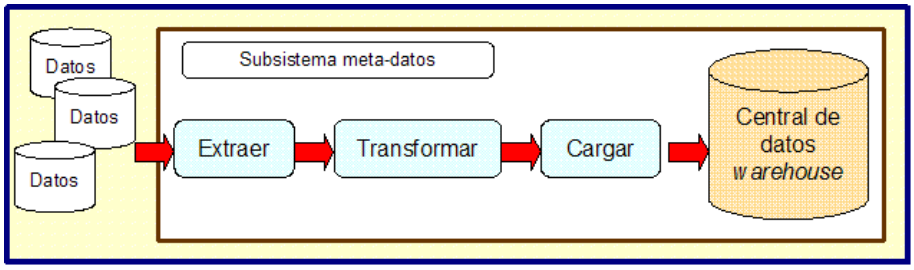

Figura 5. Data warehouse

Por otro lado, se comienzan a usar sistemas de planificación de recursos de la empresa (***Enterprise Resource Planning***, ERP). Los sistemas ERP son sistemas de gestión de empresas, típicamente implementados por un vendedor particular como un conjunto de aplicaciones integradas que cubren todas las facetas del negocio, incluyendo planificación, fabricación, ventas y marketing. La idea es usar los sistemas ERP para cubrir todas las necesidades de información de la empresa. Por desgracia, la experiencia ha demostrado que los sistemas ERP no pueden satisfacer las necesidades de información de una compañía totalmente. Si bien el código fuente está disponible para algunos de ellos, los vendedores no quieren realizar cambios específicos sobre ellos. Sin embargo, cada empresa tiene muchas particularidades que necesitan ser cubiertas sobre una base individual, además de tener que reaccionar de forma rápida para adaptarse a los cambios del mercado. Todo ello hace que se requieran modificaciones o añadidos sobre el sistema de información, y esperar las soluciones de los suministradores de los ERP no es práctico. Algunos resultados de estudios de mercado sugieren que los sistemas ERP típicamente cubren únicamente un 25-40% de las necesidades de información de una empresa (el resto se cubre con otras aplicaciones software).

La situación actual es que las empresas se encuentran con una mezcla disparatada de sistemas existentes heterogéneos de los cuales dependen sus negocios. Desgraciadamente esta mezcla de sistemas diferentes no ha sido diseñada de una manera unificada (solamente ha crecido, y continuará creciendo). La heterogeneidad consume recursos de desarrollo. En vez de crear soluciones a problemas nuevos, muchos desarrolladores emplean una considerable cantidad de tiempo y recursos trabajando sobre soluciones viejas. Y tenemos que tener en cuenta, que el número de computadores en una empresa suele ser finito.

Las aplicaciones más críticas para la mayor parte de las empresas son las aplicaciones de procesamiento de transacciones. Suelen ser aplicaciones desarrolladas a lo largo de muchos años, los desarrolladores han ido cambiando varias veces, y es difícil encontrar a

alguien que las entienda en su totalidad. Algunas de ellas todavía se ejecutan sobre sistemas *mainframe*. Las empresas son renuentes, comprensiblemente, a modificar o reemplazar dichas aplicaciones. El dilema entonces se convierte en cómo modernizarlas sin comprometer las operaciones.

1.4. Arquitectura de aplicaciones modernas

Cualquier arquitectura de una aplicación moderna debe satisfacer la necesidad de un desarrollo rápido de la solución, posiblemente mediante la composición de componentes *software.* Además, debería soportar propiedades tales como mantenibilidad, flexibilidad y escalabilidad, entre otras. La solución actual consiste en utilizar arquitecturas multi-capa, la más común contiene tres capas (ver Figura 6):

- Capa de **presentación** o interfaz de usuario. Maneja servicios tales como entrada, diálogos, y gestión de la visualización de los datos
- Capa intermedia o de **lógica del negocio**. Proporciona los servicios de la lógica del negocio que son compartidos entre aplicaciones diferentes.
- Capa de persistencia de **datos**. Proporciona una gestión de los datos y otras funcionalidades relacionadas.

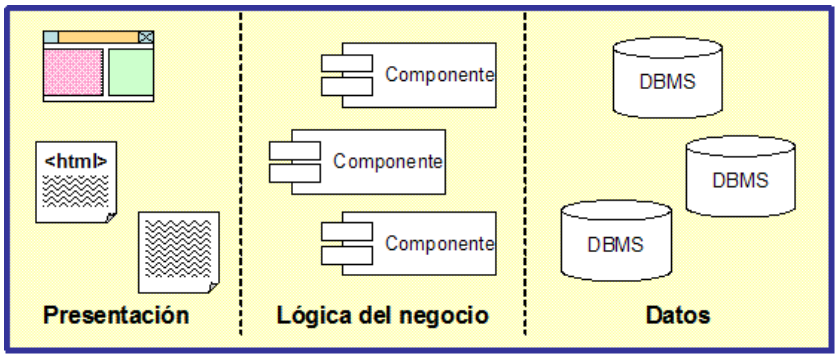

Figura 6. Arquitectura de tres capas

Cada capa, y en particular la capa de lógica del negocio, puede subdividirse en varias sub-capas dando lugar a la arquitectura lógica multi-capa. En la Figura 1.6 se pueden observar los elementos de la capa de lógica del negocio denominados **componentes**.

Un **componente** es un código ejecutable que implementa una cierta funcionalidad. Dicha funcionalidad es accesible para el resto de la arquitectura mediante uno o más *interfaces*. Los componentes están fuertemente encapsulados y ocultan sus detalles internos a sus clientes (otros componentes, métodos de objetos, procedimientos, funciones, etc.). La única forma que tiene un cliente de acceder a una componente es a través de su interfaz.

Se puede acceder a la interfaz de un componente usando un alto nivel de abstracción, de forma que no sea necesario preocuparse ni de su implementación, ni tampoco de su localización o forma de comunicación. Un ***middleware*** se ocupa de todos los detalles de la comunicación permitiendo así la interoperabilidad entre componentes (ver Figura 7).

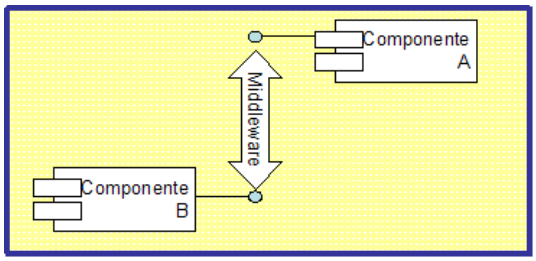

Figura 7. Interoperabilidad entre componentes

Los componentes constituyen una forma de empaquetar código ejecutable, no son ninguna técnica de implementación en sí, si bien los componentes son a menudo confundidos con objetos. Es cierto que los componentes y los objetos comparten algunos conceptos, como los interfaces y la encapsulación. Sin embargo, los componentes difieren de los objetos en que proporcionan una funcionalidad más amplia y muy poca relación con otros componentes. Cualquier cliente puede usar los servicios del componente, no importa en qué lenguaje esté escrito, ni dónde se encuentre físicamente, o cual sea su estructura interna. No es necesario que los componentes sean implementados con lenguajes orientados a objetos.

Los componentes pueden encontrarse en diferentes capas. Los componentes de la interfaz de usuario difieren de los de la capa de lógica del negocio, puesto que solamente proporcionan alguna forma de representación visual, mientras que estos últimos proporcionan fundamentalmente servicios.

Los componentes se desarrollan usando modelos de componentes. Dichos modelos proporcionan el entorno en el que se ejecutan los componentes. Modelos populares de componentes para la capa de presentación son los *JavaBeans*, o *Microsoft ActiveX*. Componentes usuales de la capa de negocio incluyen CORBA (*Common Object Request Broker Architecture*), y EJB (*Enterprise Java Beans*).

Los beneficios de una arquitectura multi-capa son obvios y numerosos: integración y reusabilidad, encapsulación, distribución, particionamiento, escalabilidad, desarrollo independiente, y desarrollo rápido, entre otros. Pero también presentan desventajas como los riesgos de seguridad, y la gestión de los componentes. Además, la arquitectura lógica tiene una arquitectura física asociada, que puede corresponderse con ésta o no (podemos ubicar cada capa lógica en una física, o agrupar varias lógicas en una sola capa física, por ejemplo para minimizar los tiempos de comunicación entre sus componentes).

Un ejemplo de arquitectura multi-capa son los **sistemas basados en web**, que presentan a menudo una capa separada de la presentación denominada capa de componentes *web*, para gestionar la lógica de la presentación de la *web*.

Con el auge de la *web*, la integración de aplicaciones adquiere un mayor significado que va más allá de la mera combinación de aplicaciones en una empresa. Los servidores de la empresa actualmente, manejan y mantienen grandes cantidades de datos y lógica del negocio. Además, debido a que la *web* facilita el fácil acceso a los servicios y la

información, se ha convertido en uno de los principales medios de comunicación. Una empresa debe ser capaz de hacer accesibles sus datos, desde empleados internos a socios externos, suministradores y fabricantes.

Ya que es un imperativo para un EIS cambiar a una arquitectura basada en *web*, las aplicaciones de la misma necesitan ser desplegadas en plataformas de aplicaciones estándar, cada vez más extensamente utilizadas. Actualmente los servidores de empresa se consideran como plataformas adecuadas para desarrollar aplicaciones *web*. Tal y como muestra la Figura 8, los servidores de aplicaciones son particularmente apropiados para las áreas B2C (*business-to-customer*) y B2B (*business-to-business*). El servidor de aplicaciones proporciona un punto de integración natural entre los sistemas de información existentes en la empresa y las aplicaciones web. El servidor de aplicaciones ayuda también en el manejo de transacciones y puede escalarse si es necesario. La plataforma J2EE es una tecnología a tener en cuenta para las empresas y vendedores de aplicaciones.

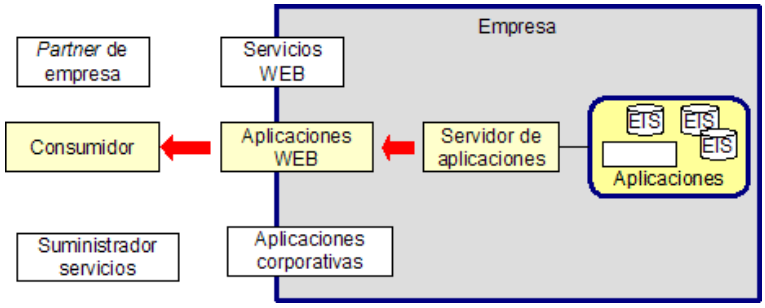

Figura 8. Integración de aplicaciones dirigida por la Web

1.5. Uso de middleware para EAI

Se denomina *middleware* al software de los servicios del sistema que se ejecuta entre la capa del sistema operativo y la capa de aplicación. Conecta dos o más aplicaciones, proporcionando interoperabilidad entre ellas. El concepto de *middleware* cobra protagonismo hoy en día, ya que todos los proyectos de integración tienen que usar una o varias soluciones diferentes *middleware*.

Usaremos el término *middleware* para denotar a aquellos productos *software* que actúan como "pegamento" entre aplicaciones, que sean distintos de simples funciones de importación y exportación de datos que pudieran formar parte de las propias aplicaciones.

Todas las formas de *middleware* son útiles para facilitar la comunicación entre diferentes aplicaciones *software*. El middleware introduce un nivel de abstracción en la arquitectura del sistema, reduciendo así su complejidad considerablemente. Por otro lado, cada producto *middleware* introduce una cierta sobrecarga en el sistema con respecto a la comunicación, lo cual puede afectar al rendimiento, la escalabilidad, y otros factores de eficiencia. Esto es importante tenerlo en cuenta, particularmente si nuestros sistemas son críticos y son usados por un gran número de clientes concurrentemente.

Los productos *middleware* comprenden una amplia variedad de tecnologías, incluyendo las siguientes:

Tecnologías de acceso a bases de datos. Proporcionan acceso a una base de datos a través de una capa de abstracción que permite cambiar el sistema de gestión de base de datos (DBMS) sin tener que modificar el código fuente de la aplicación. En otras palabras, permiten usar el mismo o similar código para acceder a diferentes fuentes de bases de datos. Las tecnologías de acceso a bases de datos difieren en la forma de los interfaces que proporcionan. Pueden ofrecer acceso a las bases de datos orientado a la función o a objetos. Los representantes más conocidos son JDBC y *Java Data Objects* (JDO) en la plataforma Java, y *Open Database Connectivity* (ODBC) y *Active Data Objects* (ADO) en la plataforma *Microsoft*.

***Middleware* orientado a mensajes**. También denominado MOM: *Message Oriented Middleware*. Es una estructura cliente/servidor que incrementa la interoperabilidad, flexibilidad y portabilidad de las aplicaciones. Permite la comunicación entre aplicaciones entre plataformas distribuidas y heterogéneas, y reduce su complejidad debido a la ocultación de los detalles de la comunicación y de las plataformas y protocolos implicados; la funcionalidad del MOM es accedida mediante APIs. Las aplicaciones pueden por lo tanto intercambiar datos sin necesidad de comprender los detalles del resto de aplicaciones, arquitecturas y plataformas implicadas.

Geralmente MOM reside tanto en la parte del cliente como en la del servidor. Proporciona comunicación asíncrona mediante el uso de colas de mensajes para almacenar los mensajes temporalmente. La comunicación puede darse incluso aunque el receptor esté temporalmente no disponible, y los mensajes pueden contener casi cualquier tipo de dato. Cada mensaje espera en la cola y es entregado tan pronto como el receptor es capaz de aceptarlo. La comunicación asíncrona tiene el inconveniente de que, ya que el servidor no bloquea a los clientes, puede continuar aceptando peticiones con el riesgo de producir una situación de sobrecarga.

La Figura 9 muestra dos aplicaciones que usan MOM. Las aplicaciones acceden a MOM a través de un API, y son responsables de construir y analizar los mensajes, pero MOM oculta todos los detalles de red y transporte.

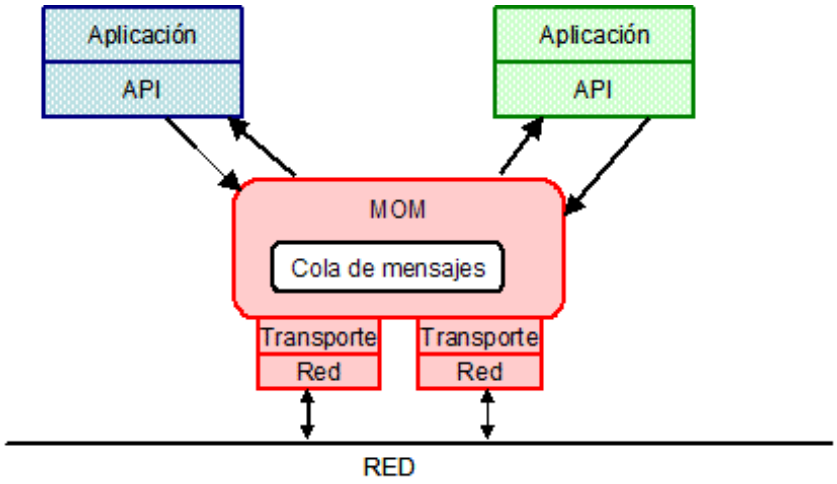

Figura 9. Comunicación de aplicaciones mediante MOM

MOM es adecuado para comunicación entre aplicaciones dirigida por eventos. También es adecuado para sistemas orientados a objetos. Los productos MOM son propietarios y están disponibles desde mediados de los 80, de forma que son incompatibles entre ellos. El uso de un único producto tiene como consecuencia la dependencia de un vendedor específico, lo que puede influenciar negativamente la flexibilidad, portabilidad, mantenibilidad e interoperabilidad de las aplicaciones. El mismo producto MOM debe poder ejecutarse sobre cada una de las plataformas que están siendo integradas. Sin embargo, no todos los productos MOM soportan todas las plataformas, sistemas operativos y protocolos. La plataforma Java proporciona formas de conseguir una relativamente alta independencia de un vendedor específico mediante una interfaz común, usada para acceder a todos los productos *middleware*: el servicio de mensajes java (JMS).

Llamadas a procedimientos remotos (RPC). Las RPC: *Remote Procedure Call* constituyen una infraestructura cliente/servidor orientada a mejorar la interoperabilidad entre las aplicaciones desde plataformas heterogéneas. Al igual que MOM, oculta los detalles de la comunicación. La principal diferencia con MOM es la forma de comunicación, mientras que MOM soporta comunicación asíncrona, RPC promueve la comunicación síncrona en forma de petición-espera, que bloquea al cliente hasta que el servidor completa la respuesta. RPC previene la sobrecarga de la red (a diferencia de MOM). Hay alguna implementación asíncrona de RPC disponible, pero son la excepción y no la regla. En la Figura 10 se muestra un esquema de dos aplicaciones que utilizan RPC.

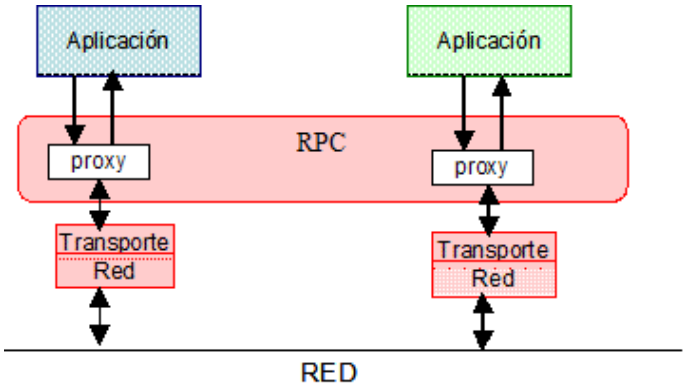

Figura 10. Comunicación de aplicaciones mediante RPC

RPC es adecuado para aplicaciones cliente/servidor el las que el cliente puede realizar una petición y esperar a que el servidor emita la respuesta antes de continuar el proceso. Además, RPC requiere que el receptor de la llamada esté disponible para aceptar la llamada remota. Si el receptor falla, la llamada remota no tiene lugar, ya que las llamadas no se almacenan temporalmente tal y como ocurre en MOM.

Monitores de procesamiento de transacciones (TP). Los TP: *Transaction Processing monitors*, son importantes en aplicaciones con requisitos críticos. Representan la primera generación de servidores de aplicaciones. Los TP se basan en el concepto de transacciones, de forma que monitorizan y coordinan las transacciones entre diferentes recursos. Si bien el nombre sugiere que esta es su única tarea, tienen otras funcionalidades adicionales: gestionar el rendimiento, y servicios de seguridad, tal y como muestra la Figura 11.

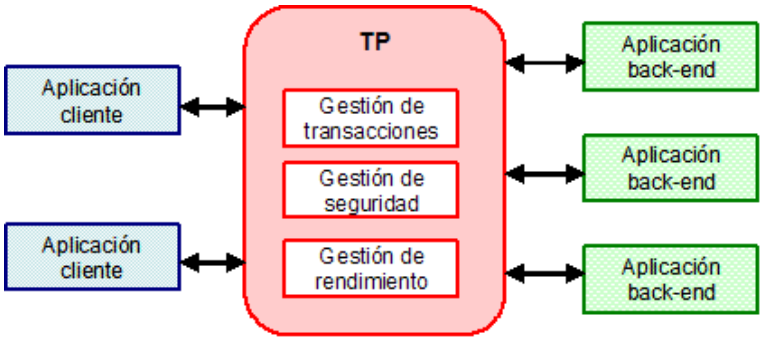

Figura 11. Comunicación de aplicaciones mediante TP

Los TP gestionan el rendimiento de las transacciones mediante técnicas de balanceo de carga y de *poolling de recursos*, de forma que pueda existir un gran número de clientes accediendo concurrentemente y de forma eficiente a los recursos de computación. Por otro lado los servicios de seguridad proporcionados por los TP permiten habilitar o deshabilitar accesos de clientes a fuentes particulares de datos.

Intermediarios de peticiones de objetos (ORB). Los ORB: *Object Request Brokers*,

gestionan y soportan la comunicación entre objetos distribuidos o componentes sin necesidad de preocuparse por los detalles de la comunicación. Los detalles de implementación de un ORB no están visibles para los componentes. De esta forma se proporciona transparencia en la localización, transparencia en el lenguaje de programación, protocolo y sistema operativo.

La comunicación se lleva a cabo a través de interfaces, y normalmente es síncrona. Proporcionan la ilusión de "localidad", haciendo que todos los componentes parezcan locales, aunque en realidad estén en otra red. Simplifican el desarrollo de forma considerable pero pueden tener una incidencia negativa en el rendimiento de la aplicación.

La Figura 12 muestra dos componentes que se comunican mediante un ORB.

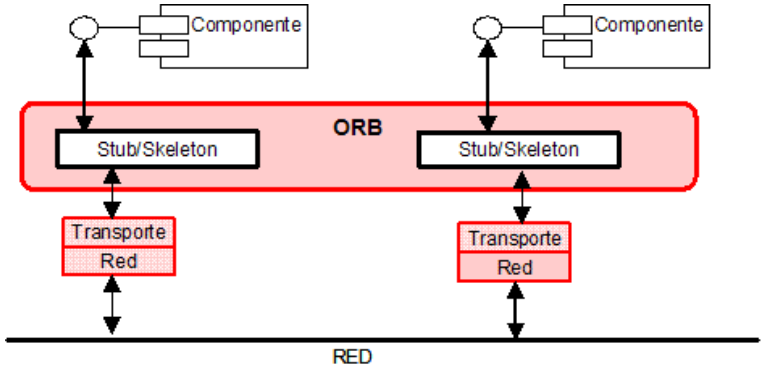

Figura 12. Comunicación de aplicaciones mediante ORB

Hay tres estándares principales para ORBs: (a) OMG CORBA, (b) Java RMI y RMI-IIOP, y (c) *Microsoft* COM/DCOM/COM+.

Hay muchos productos que cumplen el estándar ORB. RMI-IIOP. RMI-IIOP utiliza el mismo protocolo para comunicaciones entre componentes que CORBA ORB, de forma que resulta interoperable con CORBA.

Solamente existe una implementación del modelo *Microsoft*, el cual difiere de los otros dos en algunos conceptos importantes. Aún así, la interoperabilidad entre RMI-IIOP y CORBA por un lado, y el modelo de *Microsoft* por otro, es posible.

Servidores de aplicaciones. Son la forma más reciente de *middleware*. Manejan todas o la mayoría de las interacciones entre la capa del cliente y la capa de persistencia de datos. Proporcionan una colección de servicios *middleware* como los ORBs, MOM, gestión de transacciones, seguridad, balanceo de carga y gestión de recursos. Además incluyen una gestión del entorno en el que se despliegan los componentes lógicos del negocio: el contenedor.

Los servidores de aplicaciones proporcionan una plataforma excelente para integración. Independientemente de que se use para integración o para el desarrollo de aplicaciones nuevas los servidores de aplicaciones son plataformas software. Una *plataforma software*

es una combinación de tecnologías software necesarias para ejecutar aplicaciones. En este sentido los servidores de aplicaciones, o más específicamente, la plataforma software que soportan, definen la infraestructura de todas las aplicaciones desarrolladas y ejecutadas sobre ellos. Los servidores de aplicaciones son la plataforma de integración a elegir para aplicaciones desarrolladas con una arquitectura multi capa. Ademas, los servidores de aplicaciones pueden soportar una plataforma software estandarizada, abierta y generalmente aceptada.

Los aspectos más importantes a tener en cuenta en una plataforma software son:

- *Aspectos técnicos*: incluyen las tecnologías software que se incluyen en la plataforma, así como interoperabilidad, escalabilidad, portabilidad, disponibilidad, fiabilidad y seguridad, entre otras cuestiones.
- *Carácter abierto*: existen diferentes soluciones, desde plataformas cerradas asociadas a un vendedor particular, hasta plataformas totalmente abiertas, en donde cada cosa, incluido el software es de libre distribución y puede cambiarse libremente.
- *Interoperabilidad*: es crucial para considerar la adopción de una determinada plataforma, en particular la forma en que dicha plataforma regula las adiciones y modificaciones de software.
- *Coste*: probablemente es el más difícil de evaluar debido a que incluye los costes del servidor de aplicaciones, del hardware, del aprendizaje de su uso, y el coste del mantenimiento de las aplicaciones.
- *Madurez*: es un indicador de cuán estable es la plataforma. Cuanto más madura sea, más habrá sido probada en entornos reales, probando su idoneidad para aplicaciones a gran escala.

Las plataformas de integración típicamente tienen más de una tecnología *middleware*. De hecho es usual encontrar productos *middleware* de diferentes vendedores en una misma plataforma de integración. Por ejemplo, en servidores de aplicaciones que cumplen J2EE, diferentes vendedores incluyen diferentes tipos de *middleware* adicional. El servidor de aplicaciones *WebSphere* de IBM, por ejemplo, incluye el motor de mensajes IBM MQSeries, y el servidor *Bea Weblogic* incluye el monitor de transacciones *Tuxedo*.

1.6. Servicios de infraestructura necesarios para la integración

La infraestructura de integración consiste en el conjunto de tecnologías *middleware* que proporcionan los servicios necesarios para establecer la comunicación entre las aplicaciones. Para realizar una mejor selección de las tecnologías que se van a usar para la integración, en primer lugar nos centraremos en los servicios de infraestructura requeridos. Vamos a identificar dichos servicios desde una perspectiva de alto nivel y los separemos en capas horizontales y verticales. Los servicios de las capas horizontales proporcionan servicios básicos de infraestructura útiles para la mayoría de las aplicaciones existentes así como aplicaciones de nueva generación. Los servicios de la capa vertical proporcionan funcionalidades relacionadas con una tarea específica que puede extenderse a través de varios servicios de la capa horizontal.

Los servicios de las capas horizontales incluyen: (a) Comunicación; (b) *Brokering* y *routing*; (c) Lógica del negocio. Los servicios de las capas verticales son: Transacciones, Seguridad, Ciclo de vida, Nombrado, Escalabilidad, Gestión, Reglas. Los servicios mencionados se pueden apreciar en la Figura 13.

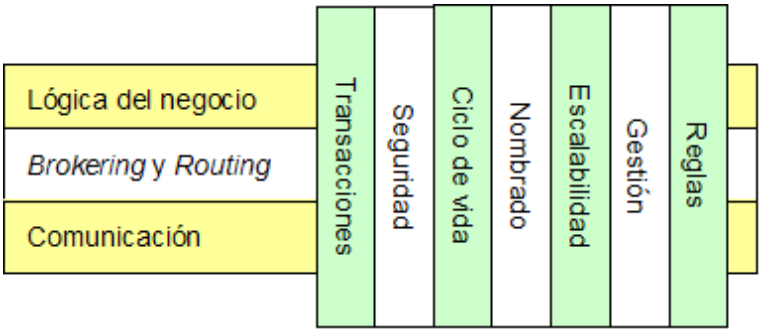

Figura 13. Relaciones entre los servicios de infraestructura de integración

Comunicación. Proporciona transparencia para el acceso a diferentes sistemas remotos y unifica la vista de los mismos. Asegura, por lo tanto, que los desarrolladores no tiene que tratar con los detalles de comunicación a bajo nivel. Debido a que la capa de comunicación no ejecuta lógica del negocio, permite una separación entre los servicios de la capa del negocio y la capa de comunicación, aunque permite comunicación entre ellas.

Las tecnologías más utilizadas son las de acceso a base de datos, como JDBC, tecnologías MOM, y tecnologías RPC. La capa de comunicación proporciona también transparencia en la localización.

Brokering y ***Routing***. Constituyen la implementación de la parte técnica de la integración. No importa que tipo de integración se use, esta capa deberá adaptar las comunicaciones entre las aplicaciones participantes en la integración de forma que todas ellas sean capaces de interoperar.

En primer lugar, esta capa tiene que proporcionar la forma de reunir los datos requeridos desde múltiple fuentes, provenientes de aplicaciones existentes y nuevas, y/o almacenamientos de datos. A esta responsabilidad se le denomina agregación, ya que los datos se obtienen de diferentes fuentes para representar un concepto del negocio, como por ejemplo una factura.

En segundo lugar, estos datos deben procesarse, de forma que esta capa tendrá que transformar y dividir los datos y mensajes en partes adecuadas para que puedan procesarse por las aplicaciones individuales.

Finalmente, esta capa debe reunir los resultados de todas las aplicaciones y presentarlos de forma consistente.

Para conseguir estos tres pasos, esta capa necesita información de meta-datos que defina las aplicaciones participantes, métodos, mensajes, e interfaces, y la secuencia de operaciones implicadas. Esta capa también proporciona formas de manejar eventos, que

serán asociados con ciertas operaciones a realizar.

No hay una única tecnología *middleware* estandarizada que proporciona todos los requerimientos necesarios para esta capa. De forma que se usarán varias tecnologías diferentes, incluyendo MOM, RPC, ORBs, y servidores de aplicaciones.

Lógica del negocio. Esta capa es la responsable de presentar un interfaz de alto nivel que permita acceder a la información del negocio para otras aplicaciones y para los usuarios, presentando los datos a los usuarios en una forma comprensible.

Actualmente, esta capa es soportada por la capa de presentación, normalmente en forma de portales personalizados.

Adicionalmente a la entrega de datos y contenidos, esta capa está conectada a menudo con tecnologías de procesamiento de datos como OLAP (*Online Analytical Processing*), *data mining* y sistemas de soporte de decisiones, entre otros. Estas fuentes analizan los datos de la empresa y proporcionan informaciones tales como estimaciones y pronósticos.

Transacciones. La infraestructura de integración debe proporcionar los medios para llevar a cabo las operaciones de la empresa de forma transaccional. Por lo tanto debe ser capaz de invocar varias operaciones sobre diferentes sistemas existentes y de nueva generación como una operación atómica que cumpla con las propiedades denominadas ACID (*Atomicity, Consistency, Isolation and Durability*).

Seguridad. Se debe proporcionar la forma de restringir el acceso al sistema. La seguridad debería incluir a las tres capas horizontales. El sistema de seguridad no debería basarse en diferentes *passwords* para diferentes aplicaciones o incluso entre partes de la misma aplicación.

Ciclo de vida. Se trata de proporcionar formas de controlar el ciclo de vida de las aplicaciones implicadas. Se debería permitir que las aplicaciones existentes fuesen reemplazadas de una en una, sin influenciar al resto de aplicaciones del sistema integrado. Hay que hacer hincapié en que este reemplazamiento debería ser posible paso a paso, cuando lo dicten las necesidades de la empresa y cuando estén disponibles los recursos suficientes. También se debe permitir que el reemplazo tenga lugar mientras el sistema permanece *on-line*. Esta última funcionalidad se consigue a menudo minimizado las dependencias entre las aplicaciones y formas específicas en las que interoperan las aplicaciones.

Nombrado. Un servicio unificado de nombres permitirá a la implementación transparencia en la localización y el reemplazo de un recurso por otro si éste es requerido. Normalmente se implementa con un producto de nombrado y directorios que permite el almacenamiento y la búsqueda de información relacionada mediante nombres. Idealmente, el servicio de nombres se debe unificar.

Escalabilidad. Es una característica que debe tenerse en mente a la hora de diseñar la infraestructura de integración. Se debe permitir el acceso a información sobre los clientes

y proporcionar acceso concurrente a las aplicaciones. Se deben incorporar soluciones que permitan extender las demandas de carga del sistema. Puede ser un problema difícil el proporcionar escalabilidad en un sistema integrado debido a que se deben tener en cuenta aplicaciones existentes que probablemente no hayan sido diseñadas para el grado de escalabilidad que deseamos conseguir. Por ello, se deben implementar algunos prototipos para probar qué niveles de rendimiento se pueden esperar. También deberían usarse herramientas de pruebas de carga que permitan simular cargas elevadas y evaluar criterios de rendimiento.

Gestión. Se deben proporcionar los mecanismos para gestionar la infraestructura de integración. El no hacerlo puede ocasionar dificultades en la fase de mantenimiento. Se debe proporcionar una gestión de versiones y configuración sencilla. Una gestión declarativa permite el acceso a cambios y actualización de parámetros sin necesidad de modificar los fuentes y volver a desplegar las soluciones. Una gestión remota permite llevar a cabo la gestión de la infraestructura de forma remota, minimizando la necesidad de entrenamiento del personal en los lugares de trabajo.

Reglas. Los servicios horizontales requieren reglas específicas para realizar la comunicación, *brokering*, *routing*, y tareas de lógica del negocio. Estas reglas no deberían estar codificadas en el "interior" de las aplicaciones, sino que deberían especificarse de una forma declarativa formando parte de la infraestructura de integración. Esto incluye definiciones, formatos de datos, transformaciones de datos y flujos, eventos, procesamiento de información y representación de la información. A menudo estas reglas se almacenan en un repositorio, lo cual proporciona un almacenamiento centralizado evitando así duplicaciones e inconsistencias.

Como hemos visto, los servicios de infraestructura pueden ser realizados por diferentes tecnologías *middeware*. Al seleccionar y reunir tecnologías diferentes hay que tener en cuenta su interoperabilidad. Alcanzar dicha interoperabilidad entre tecnologías puede ser difícil, incluso para aquellas basadas en estándares abiertos; para soluciones propietarias es todavía más difícil.

No solamente es una cuestión de si podemos alcanzar interoperabilidad, sino cuánto esfuerzo tenemos que invertir para conseguirlo. Por ello, las empresas se inclinan cada vez más hacia plataformas software que reúnen tecnologías compatibles. El uso de una plataforma software normalmente ahorra mucho trabajo adicional. Actualmente hay tres plataformas importantes:

- La plataforma Java 2, *Enterprise Edition* (J2EE)
- CORBA
- Microsoft.NET

J2EE y CORBA tienen asociadas unas especificaciones y diferentes vendedores ofrecen productos que cumplen dichas especificaciones. En este sentido son plataformas abiertas, J2EE está controlado por *Sun* y JCP (*Java Community Process*), y CORBA por OMG. Por otro lado Microsoft.NET es una plataforma propietaria y dirigida específicamente a plataformas Windows. Las tres plataformas proporcionan un conjunto de tecnologías, más

o menos apropiadas para integración.

Todos los productos EAI incluyen un componente denominado ***integration broker*** o intermediario de integración. Un intermediario de integración es una abstracción utilizada para las tecnologías que componen la infraestructura de integración. El intermediario proporciona los servicios horizontales y verticales que hemos identificado, y centraliza la gestión de dichos servicios.

El intermediario de integración se usa en todos los niveles de integración, facilitando el camino para construir las capas de integración paso a paso y reusar resultados previos, evitando cualquier dependencia de la comunicación punto a punto entre las aplicaciones.

La integración punto a punto tiene los siguientes inconvenientes:

- El número de conexiones entre aplicaciones puede llegar a ser muy grande
- Un elevado número de dependencias entre aplicaciones requiere demasiado esfuerzo de mantenimiento
- El mantenimiento de los enlaces entre las aplicaciones puede llegar a ser más costoso que el mantenimiento de las propias aplicaciones, haciendo menos obvios los beneficios de la integración

El intermediario de integración proporciona un mediador común, al que todas las aplicaciones deben conectarse, reduciendo así la multiplicidad *n-a-n* de la integración punto a punto, a *1-a-n*. Además, no solamente conecta las aplicaciones, sino que basa la integración en contratos entre aplicaciones. Estos contratos se expresan generalmente en forma de **interfaces de interoperabilidad**.

Las interfaces de interoperabilidad definen qué servicios pueden pedir las aplicaciones clientes a las aplicaciones servidor. Dichas interfaces son efectivamente contratos de larga duración que definen el acoplamiento entre aplicaciones integradas. En la medida en la que las interfaces permanecen inalterables, podemos reemplazar partes o la totalidad de aplicaciones servidor sin influenciar en nada en ninguna aplicación cliente. La Figura 14 muestra un intermediario de integración y las interfaces de interoperabilidad.

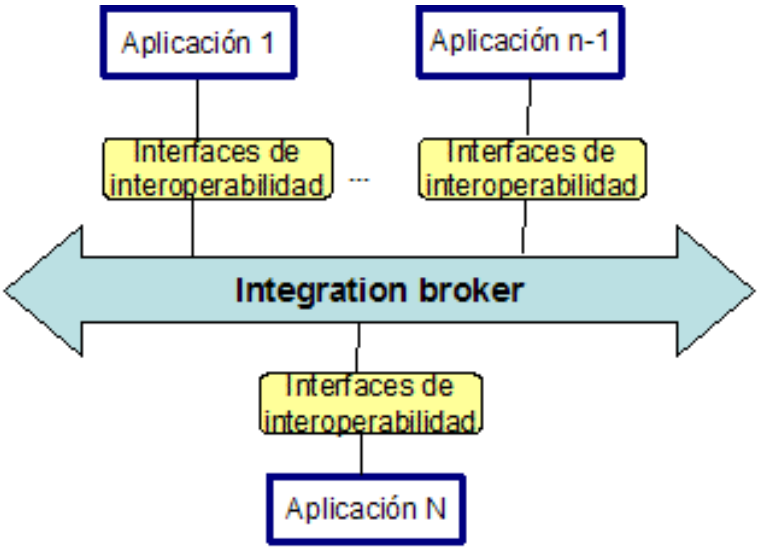

Figura 14. Intermediario de integración e interfaces de interoperabilidad

1.7. Arquitectura de integración

La arquitectura de integración especifica la estructura en su totalidad, los componentes lógicos y las relaciones lógicas entre las diferentes aplicaciones que queremos integrar.

Los dos objetivos esenciales de los sistemas de información integrados que deben ser soportados por la infraestructura de integración y por la arquitectura para un desarrollo más sencillo son:

- **Entradas de datos únicas.** Claramente, entradas de datos únicas aseguran que los datos son introducidos en el sistema únicamente una vez, garantizan la consistencia y minimizan los errores provocados por el volver a teclear los datos y la gestión de almacenamientos locales de los datos de la empresa.
- **Acceso a información con baja latencia.** También denominado acceso a datos con latencia cero. Con ello nos aseguramos que los cambios realizados en una parte del sistema de información son visibles en todas las partes relacionadas de forma inmediata (o en el menor tiempo posible).

Ambos objetivos son difíciles de alcanzar. La dificultad con respecto a las **entradas de datos únicas** la plantean la existencia de aplicaciones subsidiarias que los usuarios utilizan diariamente para proporcionar soluciones a problemas diarios de los mismos. Con respecto a la baja latencia de acceso a la información, es difícil de lograr debido a que la funcionalidad requerida para realizar las tareas de la empresa están dispersas entre aplicaciones diferentes, tanto conceptual como físicamente.

Resolver el problema de la dispersión entre aplicaciones puede lograrse de diferentes formas. Históricamente, este problema se ha tratado solamente a nivel de datos, en donde soluciones típicas incluyen la construcción de bases de datos consolidadas y *warehouses*.

Estas técnicas implican recolectar pequeñas cantidades de datos de vez en cuando y almacenándolos en la base de datos central en un formato adecuado para realizar análisis y otras tareas. Esta aproximación no soluciona el problema de la latencia cero, debido a que en el mundo real las transferencias de datos solamente se pueden realizar pocas veces a lo largo del día.

El **acceso a los datos** en tiempo real es mucho más difícil de implementar, y más teniendo en cuenta que debemos tratar con muchas fuentes de datos distribuidas. El acceso en tiempo real presenta limitaciones en cuanto a rendimiento y escalabilidad.

La forma más eficiente de acceder a los datos es a través de la lógica de la aplicación. A continuación analizaremos una arquitectura multi-capa que permite alcanzar los objetivos mencionados. Para ello nos centraremos en el concepto de **componentes virtuales**.

Un sistema de información basado en componentes virtuales es un sistema que "parece" un sistema de información desarrollado nuevamente, mientras que realmente está reusando la funcionalidad de las aplicaciones existentes.

Podemos pensar que un sistema de información basado en componentes es un reemplazamiento de todas las aplicaciones existentes, pero que no tiene los inconvenientes asociados con los reemplazamientos. Por otro lado, ya que se reusan las aplicaciones existentes no se añaden costes y no se requiere tanto tiempo adicional como en el reemplazamiento clásico de las aplicaciones.

El sistema de información basado en componentes es el objetivo último de la EAI: permite el acceso a toda la información y el uso de cualquier funcionalidad a través de componentes virtuales, con ello se consigue, además obtener información con baja latencia y garantizar que solamente necesitamos introducir la misma información en el sistema una vez.

Los **componentes virtuales** son bloques de construcción sobre los que basaremos la integración. Encapsulan los detalles y métodos que las aplicaciones existentes usan para satisfacer las peticiones realizadas sobre ellas. Por un lado, muestran la aplicación existente a través de unos interfaces, y por otro lado, se comunican con la aplicación existente utilizando las facilidades que ya tiene implementadas.

La interoperabilidad entre componentes virtuales se consigue mediante el uso de un intermediario de integración. Los componentes virtuales deberán definir los contratos, o los **interfaces de interoperabilidad**, a través de los cuales se proporcionará la funcionalidad al resto de componentes.

Los interfaces de los componentes introducirán una aproximación basada en servicios para reusar la funcionalidad, y expondrán todos ellos un conjunto de servicios normalizados que diferirán en complejidad y abstracción. En cada nivel de integración se construirán componentes virtuales más abstractos y complejos:

- A nivel de integración de datos los componentes virtuales proporcionarán únicamente acceso a los datos

- A nivel de interfaz de aplicaciones permitirán reusar además la funcionalidad de las aplicaciones existentes
- A nivel de lógica del negocio se proporcionará acceso a funcionalidades de alto nivel
- A nivel de presentación se añadirá una capa por encima haciendo que el sistema de información integrado tenga una apariencia de un sistema desarrollado nuevamente

Los componentes virtuales implementarán las funciones reusando las aplicaciones existentes. Si una función no puede implementarse únicamente con los sistemas existentes se puede añadir código nuevo. De hecho, desde el punto de vista de los clientes, los componentes virtuales no difieren de los componentes nuevos. El concepto de componentes virtuales, por tanto nos permitirán mezclar aplicaciones existentes con aplicaciones nuevas de muchas formas distintas.

Los componentes virtuales, junto con los interfaces abstractos de interoperabilidad presentan una vista de la aplicación que es independiente de la implementación: si mantenemos constante la interfaz abstracta, el resto de aplicaciones no tendrán conocimiento de los cambios realizados sobre la aplicación original.

El sistema de información integrado estará basado en una arquitectura multi-capa, que separa los roles del sistema y define las siguientes capas:

- Interfaz de usuario
- Lógica del negocio
- Persistencia de datos

Las aplicaciones existentes se situarán en la capa de persistencia de datos. En los tres primeros niveles de integración (datos, interfaz de aplicación y métodos de lógica del negocio) construiremos los componentes virtuales que proporcionarán las interfaces de las aplicaciones existentes. Estos componentes virtuales se desarrollarán en la capa de lógica del negocio.

En el nivel de integración de presentación se añadirá la nueva capa de presentación desarrollada y se desplegará sobre la capa de interfaz de usuario. Para la comunicación entre las capas utilizaremos la infraestructura de integración que proporciona el intermediario de integración. En la Figura 15 se muestra esta arquitectura multi-capa.

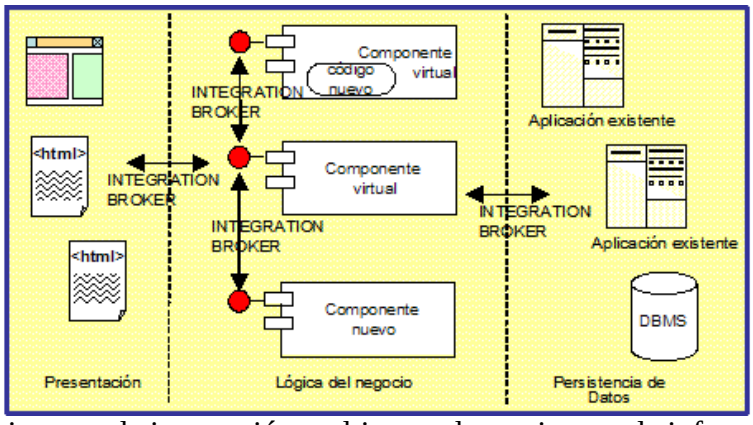

Figura 15. Arquitectura de integración multi-capa de un sistema de información basada en componentes

Para desarrollar los componentes virtuales (que son los bloques principales de construcción del sistema de información basado en componentes) usaremos una técnica denominada "envoltura" de componentes (*component wrapping*). Dicha técnica nos permite implementar la funcionalidad de los componentes virtuales mediante el uso de aplicaciones existentes.

Los componentes virtuales no presentan demasiadas dificultades para su implementación. En particular, si las aplicaciones proporcionan ya algunos API's a través de los cuales podemos acceder a su funcionalidad, ni siquiera necesitamos realizar cambios sobre las aplicaciones existentes.

Si las aplicaciones no proporcionan ningún tipo de interfaces, o si dichas interfaces no satisfacen nuestros requerimientos, entonces tendremos que desarrollarlas, o comprarlas. Para ello tendremos que modificar las aplicaciones existentes para añadir las interfaces necesarias. Denominaremos *wrapper* a la interfaz de la aplicación que vamos a añadir, mientras que a la aplicación modificada existente la denominaremos aplicación existente "envuelta" (*wrapped*).

La adición de ***wrappers*** puede realizarse de forma sencilla, especialmente en soluciones desarrolladas en la misma empresa de las que se dispone el código fuente y documentación asociada. Sin embargo, con aplicaciones en las que no se dispone del código fuente, y la documentación es limitada, realizar cambios en la aplicación puede resultar difícil. En este último caso, podemos considerar el acceder directamente a la base de datos, o incluso utilizar la interfaz de usuario para acceder a la funcionalidad. Existen técnicas como la captura de pantallas (*screen scraping*), en las que los *wrappers* acceden a la funcionalidad de la aplicación a través de la interfaz de usuario, permitiéndonos acceder a la información necesaria sin tener que modificar las aplicaciones.

En la Figura 16 mostramos la relación entre un componente virtual y un *wrapper*:

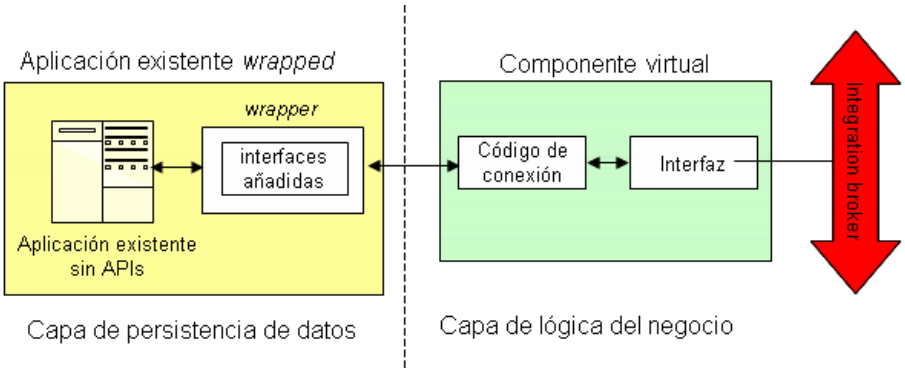

Figura 16. Componentes virtuales y wrappers

La forma en la que implementemos los *wrappers* dependerá de las tecnologías seleccionadas. Normalmente usaremos el intermediario integración para obtener la comunicación entre el componente virtual y el *wrapper*. La comunicación entre el *wapper* y la aplicación existente usará soluciones propietarias. En ocasiones integraremos el *wrapper* con la aplicación existente utilizando código fuente, comunicación entre procesos, o captura de pantallas (para así simular la entrada de datos del usuario, y leer las pantallas para obtener los resultados).

Una implementación exitosa de la capa de integración de presentación resulta en una arquitectura multi-capa tal y como se muestra en la Figura 17.

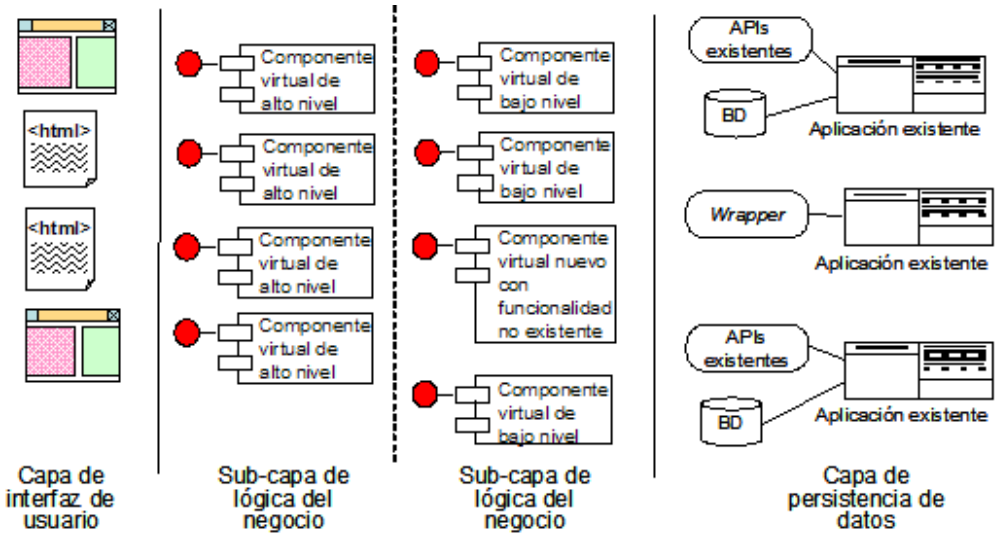

Figura 17. Arquitectura de integración

Los principales beneficios de la integración a nivel de presentación son: proceso unificado del negocio, integración de toda la aplicación, interfaz de usuario unificado, sistemas interconectados física y lógicamente, información accesible completamente sin latencia, entradas de datos únicas.

2. Arquitectura orientada a servicios (SOA)

Ya hemos comentado que muchas empresas han dedicado grandes inversiones en los recursos de sus sistemas software a lo largo de los años. Dichas empresas tienen una enorme cantidad de datos almacenados en sistemas de información (EIS) *legacy*, de forma que no resulta práctico descartar dichos sistemas existentes. Es mucho más efectivo evolucionar y mejorar los EIS e integrar las aplicaciones. ¿Pero cómo hacer ésto? La arquitectura orientada a servicios (*Service Oriented Architecture*) proporciona una solución con unos costes aceptables. Si bien la arquitectura orientada a servicios no es nueva, se está convirtiendo en el principal *framework* para la integración de entornos heterogéneos y complejos.

2.1. Razones para introducir SOA

El software de empresa está estrechamente relacionado con su organización interna, con los procesos que la forman, y con el modelo de negocio de dicha empresa. El software de empresa está condicionado tanto por las dependencias entre sus departamentos, como por las relaciones con negocios externos. Consecuentemente, una arquitectura para software de empresas debe contemplar un gran número de requerimientos diferentes. Muchos de estos requerimientos entran en conflicto, mientras que otros no están claros. En casi todos los casos, los requerimientos cambian constantemente debido al cambio permanente de los mercados, cambios en la organización de la empresa, así como en sus objetivos de negocio. Esta implicación en todos los aspectos de la empresa y del negocio es lo que hace que el software de empresa sea altamente complejo. Como dice *Dirk Krafzig*: *"El software de empresa es un animal diferente"*. El software de empresa es único en muchos aspectos, y por lo tanto, requiere medidas únicas para asegurar la eficiencia de su desarrollo y mantenimiento.

En el software de empresa, el arquitecto adquiere el rol de un influenciador y controlador externo. Tiene la responsabilidad de dirigir proyectos software individuales tanto desde el punto de vista estratégico de la organización en su totalidad, como desde el punto de vista táctico (punto de vista de la meta a alcanzar con el proyecto individual). Tiene que equilibrar diferentes requerimientos a la vez que intentar crear un orden perdurable dentro del ámbito del software de empresa. La arquitectura del software de empresa es una de las herramientas más importantes con las que cuenta el arquitecto. Éste tiene que enfrentarse constantemente con cambios y adiciones de funcionalidades que incrementan la complejidad del sistema y reducen su eficiencia. Mediante la refactorización de las soluciones actuales, los arquitectos luchan para intentar reducir la complejidad e incrementar la agilidad del sistema.

Para mejorar la eficiencia y agilidad del sistema, una arquitectura de software de empresa debe proporcionar las siguientes características:

- **Simplicidad**: A pesar de la cantidad de gente implicada, todos deben ser capaces de

comprender y gestionar la arquitectura en sus niveles respectivos (por ejemplo, especificar nuevas funcionalidades a nivel de negocio, implementarlas y mantenerlas).
- **Flexibilidad y mantenibilidad**: La arquitectura debe definir diferentes componentes que puedan reordenarse y reconfigurarse de una forma flexible. No se debe permitir que cambios locales tengan un efecto sobre el sistema global.
- **Reusabilidad**: Uno de los aspectos más importantes de la reusabilidad es el de compartir datos en tiempo real, así como el compartir funcionalidades comunes.
- **Desacoplamiento entre funcionalidad y tecnología**: La arquitectura debe hacer que la organización de la empresa sea independiente de la tecnología. En particular, la arquitectura debería evitar dependencias de productos y vendedores específicos.

Una arquitectura orientada a servicios posee las características que acabamos de comentar.

La meta última de la reusabilidad y flexibilidad proporcionada por una arquitectura orientada a servicios es el de conseguir una **empresa ágil**, en la que todos los procesos y servicios son completamente flexibles y pueden crearse, configurarse y reordenarse rápidamente cuando así sea requerido por los expertos en el negocio, sin la necesidad de personal técnico. Entre otras cosas, se facilita el dedicar más tiempo al mercado para conseguir nuevas iniciativas de negocio. Esta visión de una empresa ágil reconcilia la demanda cada vez más creciente de entornos de negocio rápidamente cambiantes, con las limitaciones de las teconologías e infraestructuras organizacionales actuales. Otras ventajas que se derivan de la agilidad de la empresa son el ahorro de costes, independencia de la tecnología, un proceso de desarrollo más eficiente, y mitigación de riesgos.

2.2. El concepto de servicio

El término "servicio" se ha venido utilizando desde hace bastante tiempo y de muchas formas diferentes. Hoy, por ejemplo, encontramos grandes compañías, como por ejemplo IBM, que promocionan el concepto de "servicios bajo demanda". Con la llegada del siglo 21, el término "servicios Web" se ha convertido extremadamente popular, si bien ha sido utilizado a menudo para hacer referencia a diferentes conceptos de computación. Por ejemplo, algunos lo utilizan para refererirse a servicios de aplicaciones que se ofrecen a los usuarios a través de la Web, como la aplicación *salesforce.com*. Otros utilizan el término "servicios Web" como módulos de aplicaciones que son accesibles para otras aplicaciones a través de Internet, mediante protocolos basados en XML. Hay que decir que servicios Web y SOA **NO** son equivalentes, sin embargo, como veremos más adelante, se pueden utilizar servicios Web para implementar una arquitectura SOA.

Para nosotros, el término servicio hará referencia a alguna actividad significativa que un programa de ordenador realiza o solicita a otro programa de ordenador. O, en términos más técnicos, **un servicio es un módulo de aplicación autocontenido que es remotamente accesible**. Los *frontends* de las aplicaciones proporcionan accesibilidad a los servicios. A veces, los términos "cliente" y "servidor" se utilizan como sinónimos de

"consumidor de un servicio" y "proveedor de un servicio", respectivamente.

Ademas, los servicios proporcionan un nivel de abstracción que oculta muchos detalles técnicos, incluyendo la localización y búsqueda del servicio. Típicamente, los servicios proporcionan funcionalidad de negocio, en lugar de funcionalidades técnicas. Otra característica de los servicios es que no se diseñan para un cliente específico, en su lugar constituyen una facilidad que contribuye a satisfacer alguna demanda pública. Es decir, proporcionan una funcionalidad que puede reutilizarse en diferentes aplicaciones. La "rentabilidad" del servicio dependerá del número de clientes diferentes que utilicen el servicio, o lo que es lo mismo, del nivel de reutilización conseguido.

Una implementación concreta de una arquitectura de servicios proporciona un acceso uniforme para todos los servicios disponibles. Así, si hacemos una analogía con la telefonía, después de que un consumidor de un servicio es "dirigido" hasta una instancia de una arquitectura de servcios, y, después de que "suene el tono de llamada", el uso de los servicios es transparente para los usuarios. Sin embargo, y como luego veremos, una arquitectura de servicios es una arquitectura en sí misma, y por lo tanto, únicamente describe la estructura y no las tecnologías concretas. Consecuentemente, las instancias de SOA podrían utilizar diferentes tecnologías en diferentes empresas.

Destacar también que los principios en los que se basa SOA son significativamente diferentes de los principios y paradigmas orientados a objetos. La diferencia clave está en que las interacciones entre los servicios se definen utilizando interfaces que están más orientadas hacia los datos que al comportamiento. Un servicio aislado puede ser implementado utilizando principios y técnicas orientadas a objetos, sin embargo, las interacciones entre estas interfaces se orientan más hacia intercambios basados en documentos. Mientras que la orientación a objetos mantiene el comportamiento "cerca" de los datos (un objeto encapsula datos y comportamiento), la orientación a servicios desacopla los datos del comportamiento. Pongamos un ejemplo sencillo: imaginemos un CD que queremos escuchar. Para poder oirlo necesitamos introducir el CD en un reproductor de CD y dicho reproductor realiza la tarea de reproducción. El reproductor de CD ofrece un servicio de reproducción de CDs. Ésto resulta muy útil ya que podemos reemplazar un reproductor de CD por otro: podemos ejecutar el mismo CD sobre un reproductor portátil o sobre un sofisticado equipo estéreo. Ambos aparatos ofrecen el mismo servicio, pero la calidad del mismo es diferente. Ahora consideremos el mismo ejemplo según la perspectiva orientada a objetos: en un estilo de programación orientado a objetos, cada CD debería contener su propio reproductor de forma no separada. Esto suena raro, pero esta es la forma en la que están construidos muchos sistemas software actuales.

El resultado de un servicio lleva consigo normalmente un cambio de estado para el proveedor, o para el cliente, o para ambos. Siguiendo con el ejemplo anterior, después de escuchar el CD en el reproductor, nuestro humor habrá cambiado, por ejemplo, de "triste" a "animado".

2.3. Definición de SOA. Elementos que la componen.

Una **arquitectura orientada a servicios** *es una arquitectura que permite la construcción de aplicaciones de negocio como un conjunto de componentes de caja negra (black-box) débilmente acoplados, que son orquestados para entregar un nivel de servicio bien definido enlazando conjuntamente diversos procesos de negocio.*

De la definición anterior, se desprende que SOA es una aproximación extensible, reutilizable y sostenible para los negocios y la tecnología, que proporciona una gran ventaja competitiva a las organizaciones que la adoptan. Resaltamos las siguientes observaciones:

- **SOA está dirigida a aplicaciones de negocio**. Existen muchas aproximaciones legítimas para las arquitecturas software, y SOA no está pensada para cualquier tipo de software. SOA está pensada para aplicaciones de negocio.
- **SOA es una arquitectura de componentes *black-box***. Siempre que sea posible, SOA oculta de forma deliberada la complejidad subyacente, y la idea de caja negra es inherente a SOA. El concepto de caja negra permite la reutilización de aplicaciones de negocio añadiendo simplemente un adaptador, sin importar cómo se hayan construido dichas aplicaciones. Los adaptadores son los que proporcionan las interfaces y hacen que SOA sea posible. Sin adaptadores no hay SOA. SOA está pensada, sobre todo, para reutilizar las aplicaciones de negocio existentes. Para hacer esto, necesitamos añadir interfaces a estas aplicaciones que nos permitan invocar directamente a las funciones que dichas aplicaciones contienen. Los adaptadores SOA proporcionan dichos interfaces.
- **Los componente SOA son débilmente acoplados**. El término "débilmente acoplado" hace referencia a la forma en la que interactúan los componentes en SOA. Un componente pasa datos a otro componente y realiza una petición. El segundo componente envía una respuesta y, si es necesario, envía datos de vuelta al componente que solicitó la petición. Se hace énfasis en la simplicidad y la autonomía. Cada componente ofrece un pequeño conjunto de servicios simples a otros componentes. Un conjunto de componentes débilmente acoplados hacen el mismo trabajo que si fuesen usados en aplicaciones con mucho acoplamiento, pero los componentes pueden combinarse y recombinarse de mútiples formas. Esto hace que la totalidad de la estructura sea mucho más flexible. También resulta mucho más sencillo crear lo que denominamos "aplicaciones compuestas" (***composite applications***), que son nuevas aplicaciones creadas a partir de las funciones de negocio de las aplicaciones existentes, y añadiendo, quizá, algún componente nuevo.
- **Los componentes software son orquestados** de forma que enlazan varios procesos de negocio para entregar un nivel de servicio bien definido. SOA crea una alineación simple de componentes que pueden, de forma colectiva, entregar un servicio de negocio muy complejo. Además, la arquitectura incluye componentes que aseguran un nivel de servicio confiable. Ejemplos de niveles de servicio pueden ser: la aplicación debe estar disponible el 99,99% del tiempo cada fin de semana desde las 6

a.m hasta las 10 p.m; en caso de un error, la aplicación debe recuperarse en 20 minutos; el tiempo de respuesta para consultas, modificaciones y creación de nuevos pedidos será de 1 segundo, y nunca peor que dos segundos, etc.

Una arquitectura orientada a servicios está basada en cuatro elementos clave: *frontend* de la aplicación, servicio, repositorio y bus de servicios (ver Figura 18).

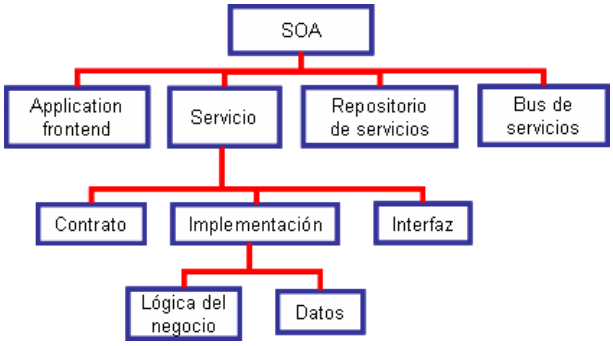

Figura 18. Elementos que componen SOA.

Si bien el ***frontend*** de la aplicación es el propietario del proceso del negocio, los servicios proporcionan la funcionalidad del negocio que los *frontends* de las aplicaciones y otros servicios pueden utilizar.

Un **servicio** consiste en: (a) una implementación que proporciona lógica del negocio y datos, (b) un contrato del servicio que especifica la funcionalidad, uso, y restricciones para el cliente (que puede ser un *frontend* de una aplicación u otro servicio), y (c) una interfaz del servicio que físicamente expone la funcionalidad.

El **repositorio de servicios** almacena los contratos del servicio de los servicios individuales de una SOA, y el **bus de servicios** interconecta los *frontends* de las aplicaciones y los servicios.

Elementos claves de una SOA

Una arquitectura orientada a servicios (SOA) es una arquitectura software basada en los conceptos clave de *frontend* de aplicaciones, servicio, repositorio de servicios, y bus de servicios. Un servicio está formado por un contrato, una o más interfaces y una implementación. Los bloques de construcción de SOA son los servicios, los cuales están débilmente acoplados para favorecer su reutilización y son altamente interoperables a través de sus contratos. Los servicios en sí mismos son "ajenos" a las interacciones requeridas a nivel de transporte para hacer posible la comunicación entre el suministrador del servicio y el consumidor del servicio.

El concepto de una SOA se centra en la definición de una infraestructura de negocio. Cuando utilizamos el término "servicio" tenemos en mente un servicio de negocio tal como *realizar una reserva de un vuelo*, o *acceder a una base de datos de clientes* de una compañía. Estos servicios proporcionan operaciones de negocio tales como *hacer una reserva*, *cancelar un billete*, u *obtener el perfil de un usuario*. A diferencia de los servicios de negocio, los servicios de infraestructura técnica, tales como un servicio de

persistencia o de transacciones, proporcionan operaciones tales como *inicio de transacción*, *actualización de datos*, o *abrir cursor*. Si bien este tipo de funcionalidad técnica es bastante útil cuando vamos a implementar una operación de negocio, tiene poca relevancia estratégica desde el punto de vista de SOA. De forma más general, la tecnología no debe tener ningún impacto sobre la estructura de alto nivel de la aplicación, ni debe provocar dependencias entre componentes. Realmente, la arquitectura orientada a servicios debe desacoplar las aplicaciones de negocio de los servicios técnicos y hacer que la empresa sea independiente de la implementación o infraestructura técnica específica.

Vamos a comentar con más detalle los elementos que componen la arquitectura SOA.

FRONTENDS de las aplicaciones

Los *frontends* de las aplicaciones son los elementos activos de la arquitectura SOA. Éstos inician y controlan todas las actividades de los sistemas corporativos. Hay diferentes tipos de *frontends*. Un *frontend* de una aplicación puede presentar una interfaz gráfica de usuario, como por ejemplo una aplicación Web o un cliente rico que interactúa directamente con los usuarios finales. Los *frontends* no tienen que interactuar necesariamente de forma directa con los usuarios finales. Los programas por lotes o los procesos de larga duración que invocan a las funcionalidades de forma periódica o como resultado de eventos específicos son también ejemplos válidos de *frontends*.

En última instancia, siempre es un *frontend* de una aplicación quién inicia un proceso de negocio y recibe los resultados. Los *frontends* de aplicaciones son similares a las capas de nivel más alto en las arquitecturas multi-capa tradicionales.

SERVICIOS

Un servicio es un componente software con un significado funcional que lo distingue de otros, y que típicamente encapsula un concepto de negocio de alto nivel. La Figura 19 muestra los elementos que forman parte de un servicio.

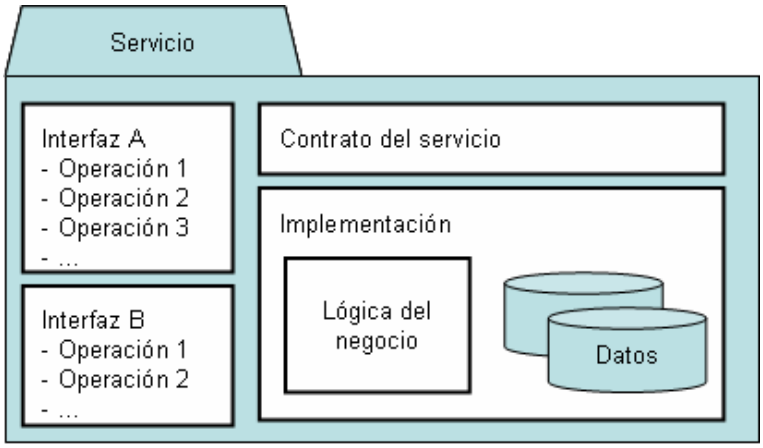

Figura 19. Elementos que componen un servicio SOA.

El **contrato** del servicio proporciona una especificación informal sobre el propósito, funcionalidad, restricciones, y uso del servicio. La forma de dicha especificación puede variar dependiendo del tipo de servicio. La definición formal de la interfaz basada en lenguajes tales como IDL y WSDL no es obligatoria. Si bien añade un beneficio significativo: proporciona una abstracción e independencia de la tecnología, incluyendo el lenguaje de programación, protocolo de red y entorno de ejecución. Es importante comprender que un contrato de un servicio proporciona más información que una especificación formal. El contrato puede imponer una determinada semántica sobre la funcionalidad o el uso de parámetros que no están sujetos a especificaciones IDL o WSDL. En realidad, muchos proyectos deben tratar con servicios que no pueden proporcionar una descripción formal de las interfaces de dichos servicios. No debemos olvidar que la tarea clave en un proyecto que pretende introducir SOA a nivel de empresa es, a menudo, no implementar nuevas funcionalidades, sino identificar módulos existentes de aplicaciones y componentes y "envolverlos" con interfaces con el nivel adecuado de funcionalidad y granularidad, haciendo que éstos estén disponibles en forma de servicios fácilmente utilizables y mejor documentados. En los casos en los que no se disponga de una descripción formal de los servicios en forma de interfaces, el servicio puede proporcionar acceso a librerías, o bien una descripción técnica detallada a nivel de protocolo de red.

La **interfaz** expone la funcionalidad del servicio a los clientes que están conectados a dicho servicio a través de la red. Si bien la descripción de la interfaz forma parte del contrato del servicio, la implementación física de la misma consiste en unos *stubs* de servicio, que se incorporan en los clientes de un servicio.

La **implementación** del servicio proporciona la lógica del proceso así como los datos requeridos. Es la realización técnica que satisface el contrato del servicio. La implementación del servicio consiste en uno o más artefactos tales como programas, datos de configuración y bases de datos.

REPOSITORIO de servicios

El repositorio de servicios proporciona facilidades para encontrar servicios y adquirir toda la información para utilizar los servicios, particularmente si estos servicios tienen que buscarse fuera del ámbito funcional y temporal del proyecto que los creó. Si bien mucha de la información requerida forma parte del contrato del servicio, el repositorio de servicio puede proporcionar información adicional, tal como la localización física, información sobre el proveedor, personas de contacto, tasas de uso, restricciones técnicas, cuestiones sobre seguridad, y niveles de servicio disponibles.

Vamos a considerar repositorios de servicios utilizados principalmente dentro de los límites de una única empresa. Los repositorios que se utilizan para integración de servicios entre empresas típicamente tienen diferentes requerimientos. En particular, aquellos repositorios públicos a través de Internet pueden requerir cuestiones legales (términos y condiciones de uso), estilo de presentación, niveles de seguridad, registro de

usuarios, suscripción a servicios, tarifas de uso, y versión utilizada).

Obviamente, un repositorio de servicios es un elemento muy útil de una SOA. Si bien no es indispensable disponer de un repositorio de servicios, éste resultará indispensable a largo plazo. Una arquitectura puede evitar el uso de un repositorio si el ámbito de un servicio es justamente un proyecto, si tiene muy pocos servicios, o si todos los proyectos son llevados a cabo por el mismo equipo de personas. En un escenario real, la mayoría de las veces habrá múltiples proyectos concurrentes, grupos de trabajo cambiantes, y una gran variedad de servicios.

Un repositorio de servicios puede ser arbitrariamente simple: en un extremo, podríamos no requerir usar ninguna tecnología. Un lote de contratos de servicio impresos localizados en una oficina y accesible por todos los proyectos es un repositorio de servicios válido. Sin embargo, hay mejores formas de proporcionar esta información, manteniendo la simplicidad del repositorio, como por ejemplo utilizar alguna base de datos propietaria que contenga datos administrativos y contratos de servicios más o menos formales para cada versión de un servicio.

En algunos casos, algunas compañías han desarrollado sus propias herramientas que automáticamente generan la descripción del servicio a partir de las definiciones formales de los servicios (por ejemplo un generador de HTML que toma un WSDL como entrada, similar al generador JavaDoc). Esto resulta particularmente útil si la definición formal del servicio se anota con información adicional sobre el servicio.

Es importante distinguir entre enlazado de servicios (*binding of services*) en tiempo de desarrollo y en tiempo de ejecución. El ***binding*** (utilizaremos los términos *binding* o enlazado indistintamente) hace referencia a la forma en la que las definiciones de los servicios y las instancias de los servicios son encontradas, incorporadas en la aplicación del cliente, y finalmente enlazadas a nivel de red.

Vamos a explicar un poco más el concepto de enlazado. Una forma sencilla de entenderlo es pensar en él en términos humanos. Cuando necesitamos hacer una llamada de teléfono a alguien, descolgamos el teléfono, marcamos el número, y esperamos mientras el teléfono suena. La persona a la que estamos llamando puede no descolgar inmediatamente, pero suponiendo que lo haga, comenzaremos formulando algún saludo, indicaremos quienes somos y por qué estamos llamando, y después de todo esto, pasaremos al asunto central de nuestra llamada.

Bien, cuando un adaptador de un componente software (recordemos que un adaptador proporciona la interfaz del componente) quiere establecer una conversación con otro, ocurren el mismo tipo de cosas que con las llamadas de teléfono. A este proceso se le denomina *binding* o enlazado, y quien establece la conexión es el *broker* del servicio (o intermediario del servicio).

Volviendo con el *binding* de servicios, si los servicios son encontrados y enlazados en **tiempo de desarrollo**, las signaturas de las operaciones del servicio son conocidas con antelación, así como el protocolo de servicio y su localización física (o al menos el

nombre exacto del servicio en un servicio de directorios). La Figura 20 describe un proceso en el que los servicios se encuentran en tiempo de desarrollo. En este caso, el desarrollador es responsable de localizar toda la información requerida en el repositorio de servicios para crear un cliente que interactúe correctamente con la instancia del servicio.

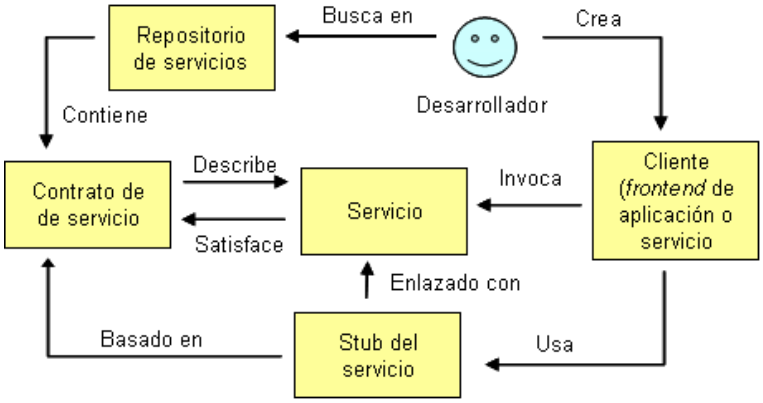

Figura 20. Descubrimiento de servicios en tiempo de desarrollo.

Si bien el enlazado en tiempo de desarrollo es un modelo bastante simple, es suficiente la mayoría de las veces. Permite a los proyectos identificar la funcionalidad que ha sido creada por proyectos anteriores y reutilizar sus servicios.

El enlazado en **tiempo de ejecución** es mucho más complejo. Podemos diferenciar entre varios niveles de enlazado:

- Búsqueda del servicio **por nombre**. Este es el caso más sencillo, y el usado más comúnmente. La definición del servicio se conoce en tiempo de desarrollo, y la lógica del cliente se desarrolla en consecuencia. El cliente puede enlazar de forma dinámica con diferentes instancias de un servicio buscandolas con nombres específicos dentro de un directorio. Por ejemplo, una aplicación cliente busca servicios de impresión con diferentes nombres, dependiendo del nombre de impresora seleccionado por el usuario.
- Búsqueda del servicio **por propiedades**. Es similar al anterior, excepto que los servicios se encuentran por propiedades. Por ejemplo, un servicio de impresión puede buscar en el repositorio impresoras a través de propiedades como la planta en la que se encuentran (p.ej. "planta ==2") y el tipo de documentos que es capaz de imprimir (p.ej. "doctype==PostScript").
- Búsqueda del servicio **basada en *reflection***. En este caso, la especificación real del servicio no se conoce en tiempo de desarrollo. Supongamos que un cliente encuentra un servicio con las propiedades del ejemplo anterior, pero con una interfaz del servicio desconocida. En este caso, en el cliente se debe implementar algún mecanismo de *reflection*, que permita al cliente descubrir de forma dinámica la semántica del servicio y el formato de peticiones válidas. Este tipo de enlazado es bastante poco usual y limitado a algunos pocos dominios de aplicaciones. Un ejemplo

de un dominio que realmente requiere un enlazado de servicios altamente dinámico es una aplicación *Bluetooth*: los clientes *Bluetooth* descubren los servicios de forma dinámica basándose en la localización y otras propiedades. Pero de nuevo, incluso en este escenario, los clientes *Bluetooth* soportan un conjunto limitado de servicios predefinidos.

En general, es recomendable tener un enlazado de servicios lo más simple posible debido a que el nivel de complejidad y riesgo crece exponencialmente con el nivel de dinamismo resultante. La búsqueda de servicios por nombre con interfaces de servicio predefinidas suele ser la opción que mejor equilibra las necesidades de flexibilidad y complejidad de implementación en la mayor parte de los casos.

BUS de servicios (ESB)

Un bus de servicios (ESB: *Enterprise Service Bus*) conecta a todos los participantes de una SOA (tanto servicios como *frontends*). El bus de servicios es similar al concepto de bus *software* definido en el contexto de CORBA. Aunque existen diferencias significativas, entre ellas la más importante es que el bus de servicios no necesariamente debe estar formado por una única tecnología, sino que puede comprender varios productos y conceptos.

El bus de servicios es el nervio central de las comunicaciones entre los servicios de una SOA. Los ESBs se diseñan para ser versátiles. Los ESBs pueden conectar con varios tipos de *middleware*, repositorios de definiciones de metadatos (como por ejemplo cómo podemos definir un número de cliente), registros (para indicar cómo localizar la información), e interfaces de cualquier cosa. Un ESB tiene no solamente tien el rol de transportador de mensajes. Las principales características de un bus de servicios son:

- **Conectividad**: el objetivo principal de un bus de servicios es del de interconectar a los participantes de una SOA. Por lo tanto debe proporcionar facilidades para permitir a los participantes de una SOA invocar las funcionalidades de los servicios.
- **Heterogeneidad de tecnología**: puesto que la realidad de las empresas se caracteriza por tecnologías heterogéneas, el bus de servicios debe ser capaz de conectar participantes basados en diferentes lenguajes de programación, sistemas operativos o soporte de ejecución. Además, normalmente encontraremos muchos productos *middleware* y protocolos de comunicación en la empresa, todos las cuales deben ser soportados por el bus de servicios.
- **Heterogeneidad de conceptos de comunicación**: es similiar a lo expuesto anteriormente pero aplicado a las comunicaciones. Obviamente, el bus de servicios debe permitir comunicaciones síncronas y asíncronas.
- **Servicios técnicos**: si bien el propósito del bus de servicios es fundamentalmente la comunicación, también debe proporcionar servicios técnicos tales como *logging*, auditorías, seguridad, transformación de mensajes o transacciones.

Podemos construir una SOA sin un ESB, pero a medida que el sistema crece y se vuelve más complejo, necesitaremos un ESB

2.4. Características de los servicios de una SOA

Los servicios son el corazón de una arquitectura SOA y deben ser: débilmente acoplados, de grano grueso (*coarse-grained*: implementan funcionalidades de alto nivel), centrados en el negocio y reutilizables.

Un servicio "bien escrito" es de **grano grueso**, lo que significa que hace referencia a que tiene un alto nivel de abstracción, y por lo tanto, su ámbito es lo suficientemente amplio como para que la gente implicada en el negocio pueda comprender el propósito del servicio incluso si tienen pocos conocimientos sobre software. Según la cantidad de colecciones de procedimientos de negocio que maneje una compañía, en la misma proporción los analistas del negocio y los profesionales del software de dicha compañia podrán compartir el conocimiento de la información de forma provechosa para ambos. Además, pueden incluir al cliente en sus las deliberaciones tempranas sobre el alcance de cada servicio, así como pueden comprender todas las implicaciones de la realización de cambios sobre un procedimiento de negocio. La facilidad de comunicaciones entre las personas es un beneficio importante de SOA y sugiere que la arquitectura se convertirá en el principio de organización primaria para los procesos de negocio.

Por otro lado, los servicios bien diseñados son **reutilizables**. Las organizaciones se benefician de la reusabilidad de dos formas al menos: primero, evitando el gasto asociado al desarrollo de un nuevo software, y segundo, incrementando la fiabilidad del software existente a lo largo del tiempo. Las compañías pueden realizar unas pruebas menos extensivas si utiliza un servicio existente en una nueva aplicación, en comparación con las pruebas necesarias requeridas para desplegar software que ha sido escrito desde cero.

A continuación discutiremos con más detalle la característica de bajo acoplamiento. El término acoplamiento hace referencia al grado en el que los componentes software dependen unos de otros. El acoplamiento puede tener lugar a diferentes niveles. Los procesos de negocio requieren un alto nivel de flexibilidad, y por lo tanto una arquitectura con un **bajo acoplamiento** para así poder reducir la complejidad total y las dependencias, y en consecuencia facilitar cambios más rápidos y con menores riesgos. En la siguiente tabla mostramos las diferencias fundamentales entre acoplamiento débil (*loose coupling*) y fuerte (*tight coupling*), considerando diferentes niveles.

Nivel	Tight coupling	Loose coupling
Acoplamiento físico	Requiere conexión física directa	Utiliza un intermediario físico
Estilo de comunicación	Síncrono	Asíncrono
Sistema de tipos	Interfaz explícita con nombres de operaciones y argumentos fuertemente tipados	Formato de mensajes flexible
Patrón de interacción	Navegación a través de complejos árboles de objetos	Centrado en los datos, mensajes autocontenidos

Control de la lógica del proceso	Centralizado	Componentes distribuidos lógicamente
Descubrimento y enlazado de servicios	Estático	Dinámico
Dependencia de la plataforma	Dependencia fuerte del sistema operativo y lenguaje de programación	Independencia del sistema operativo y lenguaje de programación

Como ya hemos comentado, una *composite application* es una aplicación compuesta a partir de servicios autónomos. Incluso si cada uno de dichos servicios ha sido construido y usado en aplicaciones anteriores que no tienen nada que ver unas con otras, éstos pueden combinarse si hay alguna razón para ello. Si estos servicios componentes pueden utilizarse fácilmente tanto conjuntamente como separados, entonces se dice que están débilmente acoplados.

Un aspecto importante sobre el acoplamiento débil es que los servicios componentes y las instrucciones básicas utilizadas para indicar cómo las "piezas" interactúan entre ellas, son separados de forma deliberada para que el servicio no contenga código relacionado con el manejo del entorno de computación. Debido a esta separación, los componentes pueden enlazarse dinámicamente en tiempo real y se comportarán como si formaran parte de una única aplicación fuertemente acoplada.

El bajo acoplamiento es posible conseguirlo debido al soporte que proporcionan varios componentes SOA, como por ejemplo el bus de servicios, el repositorio SOA, el registro SOA, y las interfaces tales como XML, SOAP, y WSDL.

Un bajo acoplamiento nos permitirá ganar en rapidez y eficiencia en varios aspectos. A continuación nombramos algunos beneficios significativos derivados de un bajo acoplamiento:

- Crear nuevas aplicaciones rápidamente utilizando servicios existentes.
- Reemplazar un servicio por otro sin tener que reescribir toda la aplicación.
- Crear aplicaciones de negocios seguras rápidamente.
- Aislar fácilmente los problemas
- Convertir los servicios software en un beneficio económico, al ofrecer su uso, previo pago, a otras organizaciones.

2.5. Capas en aplicaciones orientadas a servicios

Al igual que en cualquier aplicación distribuida, las aplicaciones orientadas a servicios son aplicaciones multicapa y tienen capas de presentación, lógica de negocio y persistencia. La Figura 21 muestra una arquitectura típica de una aplicación orientada a servicios.

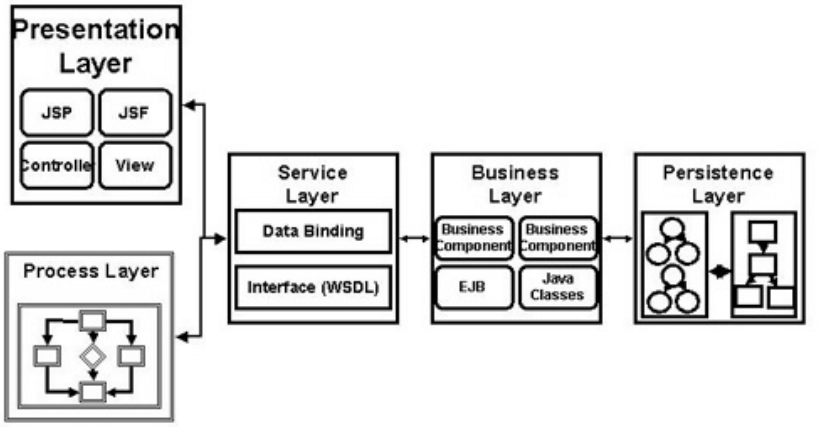

Figura 21. Capas en aplicaciones orientadas a servicios.

Las dos capas clave en una aplicación orientada a servicios son la de servicios y la de procesos de negocio.

Capa de servicios

Como ya hemos dicho, los servicios son los bloques de construcción de las aplicaciones orientadas a servicios. Los servicios son auto-contenidos, mantienen su propio estado, y proporcionan una interfaz con bajo acoplamiento.

El mayor reto cuando construimos una aplicación orientada a servicios es crear una interfaz con el nivel adecuado de abstracción. Cuando analizamos los requerimientos del negocio, tenemos que considerar cuidadosamente qué componentes software queremos construir como servicios. Generalmente, los servicios deberían proporcionar una funcionalidad con un alto nivel de abstracción (*coarse-grained*). Por ejemplo, un componente software que procesa una orden de compra es un buen candidato para ser publicado como un servicio, en contraposición con un componente que solamente actualiza un atributo de una orden de compra.

Tenemos dos posibilidades a la hora de construir un servicio: la aproximación *top-down* o *bottom-up*. La aproximación *top-down* requiere que identifiquemos y describamos los mensajes y las operaciones que proporciona nuestro servicio y a continuación implementemos dicho servicio. Esta aproximación es recomendable cuando estamos construyendo un servicio completamente nuevo, puesto que somos totalmente libres de elegir la tecnología de implementación que prefiramos. Esta aproximación también promueve servicios más interoperables, ya que podemos evitar artefactos de implementación que imposibiliten la interoperabilidad (por ejemplo, tipos de datos que pueden tener una representación interoperable).

La aproximación *bottom-up* es bastante popular debido a que nos permite reutilizar la inversión realizada en los componentes de negocio. Por ejemplo, los vendedores proporcionan las herramientas que nos permiten exponer como un servicio

procedimientos almacenados PL/SQL que comprueban si a un cliente se le puede aplicar un descuento.

El aspecto más importante de un servicio es su descripción. Cuando utilizamos servicios Web como tecnología de implementación para una SOA, el *Web Service Description Language* (WSDL) describe los mensajes, tipos y operaciones del servicio Web, que constituyen el contrato de dichos servicios.

Capa de procesos de negocio

Otra "promesa" de SOA es que podemos construir nuevas aplicaciones a partir de servicios existentes. El principal beneficio que proporciona SOA es la estandarización del modelado de procesos de negocio, a menudo referido como orquestación de servicios. Podemos construir una capa de abstracción basada en servicios Web sobre sistemas *legacy* y posteriormente beneficiarnos de este hecho para ensamblar procesos de negocio. Adicionalmente, los vendedores de plataformas SOA proporcionan herramientas y servidores para diseñar y ejecutar estos procesos de negocio. Este esfuerzo se ha materializado en un estándar de OASIS denominado *Business Process Execution Language* (BPEL); la mayoría de vendedores de plataformas se están adhiriendo a este estándar. BPEL es esencialmente un lenguaje de programación pero su representación es XML.

Si bien la sintaxis BPEL es bastante sencilla, es preferible una representación gráfica de un proceso de negocio, de esta forma nos beneficiaremos de una herramienta de diseño GUI para ensamblar nuevos procesos de negocio a partir de servicios existentes.

2.6. SOA y JBI

JBI (*Java Business Integration*) es un estándar basado en Java que aborda las cuestiones principales sobre EAI y B2B, y que está basado en los paradigmas y principios que defiende SOA. La versión 1.0 lleva el nombre de JSR (*Java Specification Request*) 208, y la versión en la que se está trabajando actualmente (2.0) se denomina JSR 312. Tanto los vendedores comerciales como los *open source* han comenzado ya a utilizar JBI como un estándar de integración en sus productos ESB (*Enterprise Service Bus*).

JBI define una arquitectura basada en *plug-ins* en la que los servicios pueden ser *plugged* en el entorno de ejecución de JBI. JBI proporciona interfaces bien definidas para los servicios que interactúan con el entorno de ejecución JBI. Los servicios necesitan exponer sus interfaces al entorno JBI para que éste pueda enrutar los mensajes hacia los servicios. El entorno de ejecución JBI actúa como un mediador entre los servicios que están desplegados en dicho entorno (ver Figura 22)

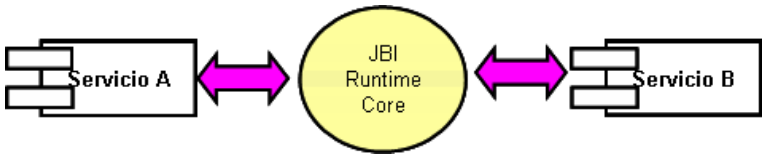

Figura 22. Entorno de ejecución JBI.

El núcleo (*core*) del entorno de ejecución JBI está formado por los siguientes componentes dentro de la misma máquina virtual Java (JVM):

- *Framework* de componentes: permite el despliegue de diferentes tipos de componentes en el entorno de ejecución JBI.
- *Router* de mensajes normalizado: permite un mecanismo estándar de intercambio de mensajes entre los servicios.
- *Framework* de gestión: está basado en JMX y permite el despliegue, gestión y monitorización de componentes en el entorno JBI

JBI adopta SOA para maximizar el desacoplamiento entre componentes, y crear una semántica de interoperación bien definida basada en mensajería estándar. JSR 208 describe las interfaces proveedoras de servicios (SPI: *Service Provider Interfaces*), que las máquinas de servicios (no definidas por JSR 208) y los *bindings* incorporan, así como el servicio de mensajes normalizado que utilizan para comunicarse entre ellos. JSR 208 tiene las siguientes ventajas de negocio:

- Es en sí misma una arquitectura orientada a servicios, y por lo tanto altamente flexible, extensible y escalable
- Las máquinas de servicios podrían implementarse en cualquier lenguaje siempre y cuando soporten la definición SPI implementada por los sistemas que cumplen JSR 208.
- Pueden añadirse nuevas máquinas en el contenedor definiendo los mensajes que utilizarán para interactuar con el resto del sistema.
- Las interfaces abiertas permiten una competencia gratuita y abierta alrededor de la implementación de estas máquinas. Esto significa que los clientes son libres de elegir la mejor solución disponible, y su código de integración puede migrarse entre las diferentes implementaciones.

Los elementos clave de un entorno JBI (mostrados en la Figura 23) son:

- Máquinas de servicios (SE: *Service Engines*), son componentes JBI que permiten una lógica de negocio *pluggable*.
- Componentes de enlazado (BC: *Binding Components*), son componentes JBI que permiten una conectividad externa *pluggable*.
- El enrutador normalizado de mensajes (NMR: *Normalized Message Router*), direcciona los mensajes normalizados desde los componentes de origen hasta sus destinatarios de acuerdo a políticas especificadas.
- El entorno de ejecución JBI (*JBI Runtime Environment*), contiene a los componentes JBI y al NMR. Debido a sus características como contenedor, a menudo se le conoce con el nombre de meta-contenedor JBI.

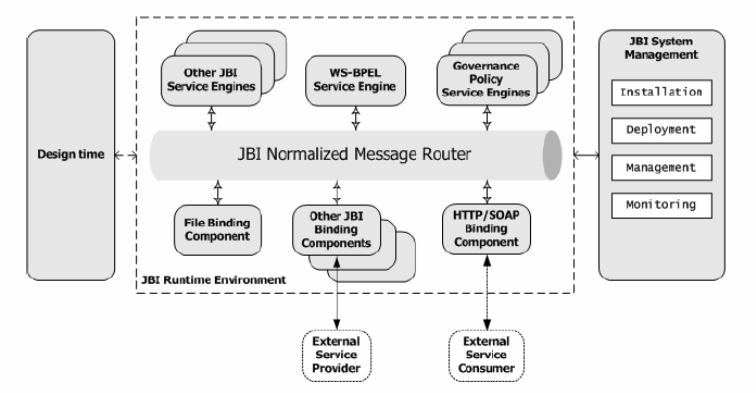

Figura 23. El entorno JBI.

Modelo de componentes JBI

JBI define dos tipos de componentes:

- **Componentes de la máquina de servicios** (SE: *Service Engine*): se trata de componentes responsables de la implementación de la lógica del negocio y otros servicios. Los componentes SE pueden ser implementados internamente utilizando varias tecnologías y principios de diseño. Los componentes SE pueden ser tan simples como un componente que proporciona servicios de bajo nivel tales como transformación y traslación de datos o algo más complejos como una instancia WS-BPEL que modela un intrincado proceso de negocio.
- **Componentes de enlazado** (*BC: Binding components*): se utilizan principalmente para proporcionar enlaces a nivel de transporte para los servicios desplegados. Los BC pueden ser de varios tipos, incluyendo: (a) los que permiten la comunicación remota con sistemas externos utilizando protocolos de transporte estándar; (b) los que permiten una invocación dentro de la máquina virtual entre dos servicios desplegados en la misma JVM; y (c) los que permiten la comunicación entre servicios utilizando ficheros de configuración WS-I (Web Services Interoperability).

El aspecto clave de JBI es el desacoplamiento de la máquina de servicios y los componentes de enlazado para que la lógica de negocio no se "infeste" con los detalles de infraestructura requeridos para invocar y consumir servicios. Esto promueve una arquitectura flexible y extensible. Tanto los componentes BC como los SE pueden actuar como proveedores y/o consumidores de servicios, así como aceptar/enviar mensajes desde/hasta el entorno de ejecución JBI.

Modelo de mensajes JBI

JBI utiliza un modelo de mensajes que desacopla los consumidores de servicios de los proveedores de servicios. El modelo de mensajes se define utilizando WSDL. WSDL se utiliza para describir las operaciones expuestas por los componentes SE y BC. WSDL también se utiliza para definir los enlaces a nivel de transporte para las operaciones abstractas de los servicios.

NMR (*Normalized Message Router*) es otro de los componente fundamentales utilizado en la arquitectura de JBI. NMR proporciona la espina dorsal construida alrededor de WSDL, quien proporciona el intercambio de mensajes débilmente acoplados entre los componentes SE y BC desplegados dentro de JBI. Se requiere que los servicios tengan interfaces, formados por un conjunto de operaciones. Cada operación está formada por uno o más mensajes. Un interfaz puede tener uno o más *bindings* a nivel de transporte.

2.7. SOA y Servicios Web

Muchos desarrolladores piensan a menudo que los servicios Web y SOA son sinónimos. Muchos también piensan que no es posible construir aplicaciones orientadas a servicios sin utilizar servicios Web. Para clarificar las diferencias diremos que SOA es un principio de diseño, mientras que los servicios Web son una tecnología de implementación. Por lo tanto, podemos construir una SOA utilizando otras tecnologías tradicionales, como por ejemplo RMI.

Tenemos que destacar que el precursor de SOA es *Jini*: un entorno de computación distribuida basado en Java y desarrollado por *Sun* a finales de los 90, en la que los dispositivos (como por ejemplo impresoras, portátiles y PDAs) pueden ser *plugged* en una red y automáticamente ofrecer sus servicios y hacer uso de otros servicios en dicha red. Posteriormente los servicios Web utilizan estándares independientes de la plataforma tales como HTTP, XML, SOAP, y UDDI, permitiendo la interoperabilidad entre tecnologías heterogéneas tales como J2EE y .NET.

En 2003, SOA entra al fin por completo en el mundo de las tecnologías de la información empresariales, a través de los servicios Web, y gracias a:

- Al contrario que CORBA y DCE, los estándares de servicios web no tienen detractores entre los fabricantes.
- Los servicios Web tienen flexibilidad para soportar aplicaciones multicanal.
- La capacidad de SOAP de atravesar los *firewalls*, aprovechando la ubicuidad del HTTP.
- El soporte de servicios Web en servidores de aplicaciones que albergan lógica empresarial.
- Los ESBs, que combinan servicios Web con *middleware* orientado a mensajes (MOM), más algunas capacidades de transformación y enrutado.

SOA está emergiendo como el principal marco de integración en entornos de computación complejos y heterogéneos. Los intentos anteriores no permiten soluciones interoperables abiertas, ya que recaen sobre APIs propietarios y requieren un alto nivel de coordinación entre grupos de trabajo. SOA puede ayudar a las organizaciones a modernizar sus procesos para que sus negocios sean más eficientes, así como adaptarlos a las necesidades de cambio y ser más competitivos, gracias al concepto de servicio. *eBay*, por ejemplo, ha "abierto" los APIs de sus servicios Web de sus subastas *online*. El objetivo es permitir a los desarrolladores ganar dinero a través de la plataforma *eBay*.

Utilizando los nuevos APIs, los desarrolladores pueden construir aplicaciones cliente que enlacen con el sitio *online* y permitir que las aplicaciones acepten ítems que se pondrán a la venta. Tales aplicaciones suelen interesar a vendedores, ya que los compradores todavía utilizan *ebay.com* para pujar sobre los items. Este tipo de estrategia, por lo tanto, incrementará la base de clientes de *eBay*.

Aunque una SOA se podría implementar sin servicios Web, siendo realistas, cualquier proyecto SOA realizado en los últimos años está plagado de ellos. Es más, la mayoría de los clientes empiezan a practicar con SOA creando un servicio Web, muchas veces sobre lógica de negocio ya existente.

Una de las principales ventajas de implementar una SOA con servicios Web es que los servicios Web están muy extendidos y constituyen una plataforma sencilla y sobre todo neutral.

La arquitectura básica de los servicios Web (mostrada en la Figura 24), consiste en especificaciones (SOAP, WSDL, y UDDI) que soportan la interacción de un servicio Web que realiza una solicitud (servicio Web cliente) con un servicio Web que suministra el servicio requerido y con un registro de servicios (directorio de servicios Web). El suministrador publica una descripción WSDL de su servicio Web, y el solicitante accede a dicha descripción utilizando UDDI u otro tipo de registro, y solicia la ejecución del servicio del suministrador enviandole un mensaje SOAP.

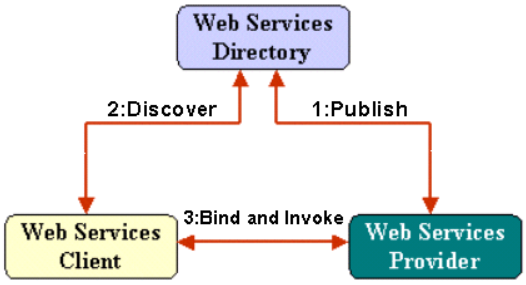

Figura 24. Arquitectura básica de servicios Web.

La combinación de servicios Web y SOA proporciona una integración rápida. Consideremos un ejemplo con tres aplicaciones de bases de datos de la industria de las finanzas que soportan préstamos bancarios, operaciones comerciales y operaciones de inversiones bancarias. Supongamos que dichas aplicaciones se han desarrollado utilizando una arquitectura clásica de tres capas, separando la lógica de la presentación, la lógica del negocio, y la lógica de la base de datos.

Tal y como se muestra en la Figura 25, es posible reutilizar una aplicación tradicional de tres capas como una aplicación orientada a servicios creando servicios a nivel de la capa de lógica de negocio e integrando dicha aplicación con otras aplicaciones utilizando el bus de servicios. Otro beneficio de la orientación a servicios es que es más fácil separar la lógica de presentación de la lógica del negocio cuando la capa de lógica de negocio está preparada para soportar servicios. También es más fácil conectar varios tipos de GUIs y

dispositivos móviles con la aplicación cuando la capa de lógica de negocio soporta servicios. En lugar de ejecutar la lógica de la capa presentación como una interfaz altamente acoplada sobre el mismo servidor, la lógica de presentación puede situarse sobre un dispositivo separado, y la comunicación con la aplicación puede realizarse a través del bus de servicios.

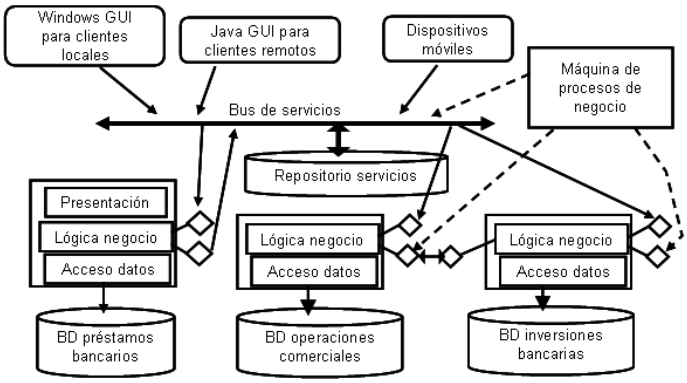

Figura 25. Diseño que facilita una integración orientada a servicios.

Otro aspecto importante es que las aplicaciones pueden intercambiar datos más fácilmente utilizando un servicio Web definido en la capa de lógica de negocio que utilizando otra tecnología de integración diferente debido a que los servicios Web representan un estándar común entre todos los tipos de software. XML puede utilizarse independientemente de la defínción de los tipos de datos y estructuras. Finalmente, el desarrollo de puntos de entrada orientados a servicios en la capa de lógica de negocio permiten a una máquina de gestión de procesos de negocio llevar a cabo un flujo automático de ejecución a través de los múltiples servicios.