FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: Las tecnologías SOA y ESB como herramientas integradoras para el acceso unificado a servicios colaborativos heterogéneos

AUTORES: Boccalari Cristian Luján

DIRECTOR: Prof. Javier Díaz

CODIRECTOR: Prof. Claudia Queiruga

CARRERA: Licenciatura en Informática

Resumen

El marco teórico de esta tesis refiere a cuestiones relacionadas con la integración de aplicaciones en general, y de herramientas colaborativas asincrónicas en particular, especialmente aquellas usadas por los usuarios y desarrolladores de las comunidades “alrededor” de proyectos FLOSS (Free/Libre Open Source Software).

Presenta la importancia del conocimiento de patrones de diseño en el ámbito de la integración de aplicaciones, del uso e implementación de SOA, ESB/JBI como elementos importantes al momento de brindar una solución que sea flexible, estándar, y escalable en el tiempo.

Entre las alternativas existentes para la construcción de la plataforma de integración, la más destacada comprende una combinación de una arquitectura escalable, orientada a servicios (SOA), un componente de middleware estándar, flexible, con soporte a múltiples protocolos, tecnologías y formato de datos heterogéneos (ESB/JBI), y fundamentalmente mediante el uso de tecnologías, librerías y frameworks con licencias libres, que permitan su estudio, modificación y adaptación a requerimientos específicos.

Líneas de Investigación

He investigado respecto a las cuestiones relacionadas con la integración de aplicaciones, tecnologías y patrones de integración en general, y en particular sobre SOA, ESB y JBI como una combinación adecuada para la implementación de una plataforma de integración estándar, flexible y escalable.

Para la implementación del ESB he estudiado e investigado numerosas herramientas y librerías de software basadas en Java, con licencias libres u open source, especialmente aquellas apoyadas por la fundación Apache.

Trabajos Realizados

Este informe está acompañado de la implementación de un ESB y un motor de búsqueda full text, utilizando únicamente herramientas y librerías basadas en Java, y con licencias libres/open source.

Estos dos componentes son esenciales en la construcción de una plataforma de integración flexible y escalable, que permita la integración y homogenización de los resultados de búsquedas de información en herramientas colaborativas asincrónicas, especialmente en aquellas usadas en proyectos FLOSS.

Conclusiones

La construcción de una arquitectura de integración orientada a servicios raramente involucra la creación "desde cero" de los servicios involucrados en la integración de aplicaciones.

Por lo tanto la consideración del uso, reuso y adaptación de determinadas herramientas y librerías FLOSS para la implementación del ESB, del motor de búsqueda full text, servicios y componentes necesarios, tiene sus razones principalmente en la posibilidad que ofrecen de poder estudiar, modificar, adaptar, reutilizar y optimizar el funcionamiento originalmente provisto, gracias a la disponibilidad del código fuente.

Sin el uso de herramientas y librerías FLOSS no hubiese sido una tarea sencilla, e incluso siquiera posible, el diseño e implementación de una plataforma de integración orientada a servicios.

Trabajos Futuros

La información obtenida desde los repositorios asincrónicos de proyectos de software libre utilizando la plataforma de integración implementada, puede ser clasificada, organizada, y filtrada en base a técnicas de data mining.

Es interesante también la posibilidad de escalabilidad que tiene la plataforma de integración al permitir la implementación de un sistema de clustering de contenedores ESBs, que posibilite el balanceo de carga y tolerancia a fallos.

Fecha de la presentación: Junio 2010

A mi familia y amistades...

Índice General

Capítulo 3

Capítulo 6

Capítulo 7

Capítulo 8

Capítulo 9

Resumen

El marco teórico de esta tesis refiere a cuestiones relacionadas con la **integración de aplicaciones** en general, y de herramientas colaborativas asincrónicas en particular, en proyectos **FLOSS** (Free/Libre Open Source Software)[1]. Presenta la importancia del conocimiento de **patrones de diseño** en el ámbito de la integración de aplicaciones, del uso e implementación de **SOA**, **ESB/JBI** como elementos importantes al momento de brindar una solución flexible, estándar y escalable en el tiempo.

FLOSS como modelo innovador de desarrollo es apoyado por una comunidad de usuarios/desarrolladores generalmente descentralizada, donde la participación y el grado de contribución usualmente es voluntario y se sustenta en que todos y cada uno de los miembros de la comunidad puedan cosechar beneficios derivados de su participación en la comunidad.

Los participantes de la **comunidad** generalmente son especialistas en el uso de tecnologías de la información y la comunicación y apoyados por una red de interacciones construyen software mediante la producción entre pares y el uso intensivo de herramientas colaborativas, accesibles a través de una red, que facilitan la tarea de organizar el trabajo cooperativo y descentralizado de multitud de personas.

Sin embargo la mera inclusión de **herramientas colaborativas asincrónicas** en proyectos de software no significa una mejora, ayuda o simplificación del desarrollo de software. Es probable que la información relevante para la toma de decisiones o solución de errores esté **dispersa** en múltiples fuentes de información y su búsqueda y recuperación no sea trivial.

La **integración de aplicaciones** permite conectar múltiples sistemas, generalmente heterogéneos, facilitando que éstos cooperen y provean funcionalidad unificada. De esta manera es posible que los resultados de las búsquedas de información en herramientas colaborativas asincrónicas diversas puedan ser integrados y estructurados en un formato uniforme, accesible a través de diferentes protocolos de comunicación y formatos de datos.

Entre las alternativas existentes para la construcción de una plataforma de integración, la más destacada comprende una combinación de una arquitectura escalable, orientada a servicios (**SOA**), un componente de middleware estándar, flexible, con soporte a múltiples protocolos, tecnologías y formato de datos

1 **FLOSS**: http://es.wikipedia.org/wiki/FOSS_-_FLOSS

heterogéneos (**ESB/JBI**) y fundamentalmente mediante el uso de tecnologías, librerías y frameworks con **licencias libres**, que permitan su estudio, modificación y adaptación a requerimientos específicos.

La implementación de la plataforma de integración, denominada ***jESBihca (Java ESB para la Integración de Herramientas Colaborativas Asincrónicas)***, desarrollada en Java, está basada fundamentalmente en una implementación del estándar JBI (*Java Business Integration*), Apache ServiceMix, y en librerías y frameworks adicionales, mayormente surgidos en la fundación Apache. Todas las herramientas utilizadas tienen licencias libres u open source.

jESBihca es desplegada para **integrar** y **homogeneizar** los resultados de búsquedas federalizada a las herramientas colaborativas (una WIKI, un bug tracker, un manejador de versiones de código fuente y el sistema de archivos) utilizadas por un **hipotético** proyecto FLOSS, denominado ***jIntegra***. Se plantea un **caso de estudio** sencillo con escenarios de búsqueda a las herramientas colaborativas de *jIntegra* basados en tres tipos de consultas posibles: a través de texto plano (no estructurado), de documentos XML (estructurado y con un formato preacordado) y de mensajes SOAP (estructurado y con un formato estándar).

Palabras claves

Integración de aplicaciones, Patrones de diseño en Integración, Arquitecturas Orientadas a Servicios, Enterprise Service Bus, Java Business Integration, Free/Libre Open Source Software

Reconocimientos

Estas, quizás, sean las líneas del informe más difíciles que me toca escribir, no por no creer saber que quiero expresar, y a quienes dedicárselas, sino por miedo a olvidarme de alguien que debiera nombrar en este momento. Es así que mis reconocimientos serán más bien generales.

En primer lugar debo **agradecer infinitamente**, y de manera prioritaria, a mis **familiares**, mi **pareja** y **amigos** todos, que son las personas especiales con las cuales comparto y he compartido los mejores momentos de mi vida, que enriquecen mi presente y me animan constantemente a continuar mejorando como persona.

También debo un reconocimiento sincero y especial a los profesores **Claudia Queiruga** y **Javier Díaz**, directores de esta tesis, quienes desde un principio me dieron absoluta libertad de acción y elección de los temas incluídos en este trabajo, y que desde su atenta mirada, su observación crítica y enriquecedora, me guiaron en la ardua tarea de concretar este trabajo final.

Es mi deber, y así lo siento, hacer extensivo mi reconocimiento y admiración hacia todos los involucrados en proyectos de **software libre**, cualquiera sea su nivel o espacio de participación, pues cotidianamente sus pequeñas o grandes acciones ayudan a consolidar un modelo de construir software innovador, solidario, basado en principios que pregonan la horizontalidad, y estimulan la participación de todos, sin más que la firme convicción (quizás utopía) de sostener que el **conocimiento** debe ser **libre** y debe estar a disposición de todos para su aprovechamiento.

Finalmente, considero necesario e importante reconocer en este momento el rol que han cumplido y cumplen las **universidades de carácter público y gratuito**, en la difícil misión de formar profesionales y personas de bien. De no ser por los docentes, alumnos y demás personas que han luchado y luchan cotidianamente para sostener una educación universitaria de calidad, que continue siendo pública y fundamentalmente gratuita, quizás muchos de mis compañeros de estudio, otros tantos amigos, conocidos, e inclusive yo, no hubiesemos tenido probablemente la posibilidad de acceder a estudiar una carrera universitaria.

A modo de síntesis, para remarcar la satisfacción de haberme formado, desde el jardín de infantes, en instituciones educativas públicas, incluída, por supuesto,

la Universidad Nacional del La Plata, transcribo un pequeño párrafo extraído de la famosa proclama, conocida como Manifiesto Liminar, redactada en 1918 por alumnos reformistas de la Universidad de Córdoba, punto de inflexión en la transformación hacia una universidad tal cual la conocemos hoy, deseando que ese momento tan relevante en nuestra historia educativa nos guíe para **no claudicar jamás** en la lucha cotidiana, dura, usualmente desigual, frente a quienes han pretendido, pretenden y seguramente pretenderán apoderarse y transformar **nuestra universidad pública** en una de carácter elitista, privada, y para unos pocos privilegiados.

Termino entonces este humilde reconocimiento a la universidad pública con el párrafo prometido.

"La juventud [...] es desinteresada, es pura. No ha tenido tiempo aún de contaminarse. Ante los jóvenes no se hace mérito adulando o comprando. En adelante, sólo podrán ser maestros en la futura república universitaria los verdaderos constructores de almas, los creadores de verdad, de belleza y de bien [...] si no existe una vinculación espiritual entre el que enseña y el que aprende, toda enseñanza es hostil y por consiguiente infecunda. Toda la educación es una larga obra de amor a los que aprenden..."

¡¡¡ Nuevamente, Gracias a Todos !!!

“Si tienes una manzana y yo tengo una manzana y las intercambiamos entonces seguiremos teniendo una manzana cada uno. Pero si tienes una idea y yo tengo una idea e intercambiamos estas ideas, entonces cada uno de nosotros tendrá dos ideas.. .” [2]

2 George Bernard Shaw

Prefacio

La organización general de este trabajo está dividida en dos partes principales.

La parte I, que incluye los capítulos 1 a 5, introduce los fundamentos teóricos que darán sustento a la implementación de la plataforma de integración ***jESBihca***, que acompaña este trabajo. La parte II, que abarca los capítulos 6 a 10, está dedicada exclusivamente a presentar el caso de estudio y sus respectivos escenarios de prueba, las tecnologías involucradas, y aspectos meramente técnicos de la implementación de la plataforma de integración.

La parte I se organizó de manera de analizar, a los fines de este trabajo, en primer lugar los temas más generales y resaltar aquella temática más específica que será objeto de análisis de los capítulos siguientes.

El capítulo 1 brinda una introducción general a FLOSS y su modelo de desarrollo. FLOSS y las comunidades "alrededor" de este tipo de proyectos, conforman uno de los disparadores que da origen a la motivación y posterior investigación de los temas incorporados en este trabajo de grado. Aquí sólo se analiza FLOSS y sus particularidades desde un punto de vista general, para luego, en el capítulo 6, correspondiente a la parte II, orientar el estudio a un proyecto FLOSS en particular, analizando características útiles que ayuden a justificar y comprender la motivación del caso de estudio planteado.

El capítulo 2 está enfocado en la presentación y análisis de la problemática de integración de aplicaciones en general. Presenta las diferentes estrategias y estilos de integración existentes, haciendo especial hincapié en uno de los estilos de integración, mensajería, importante en el análisis posterior de los ESBs. Este capítulo analiza luego las principales tecnologías de integración existentes, entre las cuales se encuentran los servicios web y ESB (*Enterprise service bus*). Los servicios web serán posteriormente analizados en el capítulo sobre SOA (Service Oriented Architecture), mientras que ESB tiene un capítulo completo dedicado a su estudio. Finalmente, el capítulo 2 realiza una introducción a los principales patrones de diseño en integración de aplicaciones. Esta descripción teórica de los patrones de integración servirá de sustento del capítulo 9 de la parte II, donde se analizan los servicios de jESBihca que implementan algunos de los patrones de integración más conocidos.

El capítulo 3 está dedicado completamente al análisis de la evolución y características generales de las arquitecturas orientadas a servicios (SOA). Este

capítulo es el punto de inflexión entre los fundamentos teóricos y la implementación práctica del caso de estudio, ya que nos permite ir acercándonos a las tecnologías y herramientas necesarias y utilizadas en la implementación de la plataforma de integración. Básicamente se introducen las tecnologías fundacionales de SOA, su evolución, relación con servicios web, otras tecnologías, y especialmente con ESB, analizado y descripto como un componente esencial en toda infraestructura orientada a servicios.

El capítulo 4 está enfocado únicamente en el análisis y estudio de los ESBs, considerado previamente como un componente esencial en la infraestructura de una arquitectura orientada a servicios. Básicamente se presentan, de acuerdo a autores especialistas en el área, las características deseables que debería tener un ESB, su relación con otras tecnologías analizadas y presentadas en capítulos anteriores y especialmente su importancia y ventajas tecnológicas para la resolución del problema de integración de aplicaciones heterogéneas.

Finalmente el capítulo 5, está destinado al estudio de JBI (*Java Business Integration*), especificación promovida por la JCP (Java Community Process) que estandariza la implementación de servicios útiles para el reuso e integración de aplicaciones.

Como anteriormente se describió, el capítulo 4 analiza las características deseables, componentes y aspectos técnicos que debería tener una implementación exitosa de un ESB. Sin embargo, tal capítulo sólo realiza un estudio teórico y conceptual de los ESBs. Para alcanzar la implementación de un ESB que permita construir la plataforma de integración pretendida en la parte II de este informe, es menester introducir y analizar la especificación JBI, la cual fue seleccionada por adecuarse a ciertos requerimientos planteados por el autor de esta tesis.

El capítulo 6 indica el comienzo de la parte II, destinada al análisis y presentación de la implementación del ESB, basado en tecnologías con licencias libres/open source. Este capítulo realiza una breve introducción a las características generales de las comunidades de usuarios/desarrolladores que se forman “alrededor” de los principales proyectos FLOSS. En particular, el análisis se realiza considerando mi experiencia personal al momento de participar en la comunidad de usuarios pertenecientes al proyecto Apache ServiceMix, elegido por ser una herramienta fundamental en la implementación de **jESBihca**.

La idea de los capítulos 6 y 7 es introducir las características especiales del funcionamiento de una comunidad de usuarios descentralizada que colabora en el sostenimiento de un proyecto libre, en particular para justificar el por qué de la necesidad del uso de herramientas colaborativas asincrónicas para la organización y coordinación del trabajo descentralizado.

Se presentan los tres escenarios de búsqueda elegidos para probar la funcionalidad de la plataforma de integración **jESBihca**, mostrando los componentes y servicios involucrados en su ejecución.

Los capítulos siguientes, 8, 9 y 10, están dedicados al análisis detallado de la implementación de los distintos componentes y servicios que conforman **jESBihca**, junto a las distintas librerías en cada caso.

Parte I - Fundamentos

- ↙ **Motivación e Introducción**
- ↙ **Integración de aplicaciones**
- ↙ **Arquitecturas orientadas a servicios (SOA)**
- ↙ **Enterprise Service Bus (ESB)**
- ↙ **Java Business Integration (JBI)**

Capítulo 1

Motivación e Introducción

1.1. Motivación

La principal motivación en la elección del tema del trabajo de grado fue fundamentalmente mi necesidad particular al momento de **buscar** información para **conocer** o **involucrarme** en proyectos **FLOSS**. La información buscada estaba dispersa en diferentes lugares, lo cual dificultaba su búsqueda y acceso.

Esta problemática también surgió en ciertas experiencias laborales, siendo un trabajador independiente, principalmente en diversos proyectos informáticos, usualmente con licencias libres u *open source*, cooperando con expertos en distintas áreas IT, y geográficamente distantes.

El crecimiento en la cantidad y tamaño de los proyectos, fue el principal motivo para incorporar y utilizar diversas herramientas asincrónicas de ayuda al trabajo colaborativo y descentralizado, que permitiesen organizar y centralizar ciertas tareas, como así también documentar ciertos procesos. No sólo aumentaban los proyectos, la mayoría interrelacionados, sino también el número de herramientas de soporte al trabajo descentralizado. Surgieron nuevos problemas, externos al desarrollo específico del software, y relacionados principalmente con la administración, búsqueda y clasificación de los documentos producidos y almacenados por las diversas herramientas incorporadas.

Otra gran dificultad encontrada fue la necesidad de adaptar ciertas funcionalidades de algunas de las herramientas a cuestiones propias de cada proyecto. El uso de herramientas privativas (código cerrado) hizo imposible encontrar una solución “limpia” y adecuada a dichos problemas, con lo cual se optó por el reemplazo de las herramientas por otras con licencias FLOSS.

El disponer del código fuente de las aplicaciones facilitó la tarea de adaptarlas a requerimientos específicos de cada proyecto.

Ese período de trabajador independiente me dejó muchas inquietudes e intereses relacionados específicamente con el uso de herramientas colaborativas de ayuda al proceso de desarrollo de software, el uso y adaptación de herramientas y librerías con licencias libre/open source y, principalmente, metodologías de integración de aplicaciones heterogéneas.

1.2. ¿Qué es FLOSS[3]?

Puede resultar sorprendente para aquellos que no conozcan los orígenes del software, que éste, como entidad separada del hardware, es una concepción relativamente reciente. En los comienzos, y debido a la complejidad, costos de desarrollo y limitado poder de procesamiento de las primeras computadoras, el código fuente era compartido libremente entre los usuarios, de una manera colaborativa, y que condujo al surgimiento de diferentes grupos de usuarios, especialmente en el ámbito académico.

En los 70's se produce la "separación" entre el hardware y software, principalmente con el surgimiento de las computadoras personales, siendo este último empaquetado y comercializado separado del hardware [1]

A finales de la misma década algunas compañías de software comenzaron a imponer restricciones a los usuarios a partir del uso de licencias privativas.

Richard Stallman funda en 1983 la Free Software Foundation (FSF)[4] a fin de lograr la preservación de las libertades de los usuarios de usar, estudiar, modificar y distribuir el software

La FSF introduce la definición formal de "software libre" como áquel que brinda 4 libertades a los usuarios:

- La libertad de ejecutar el programa para cualquier propósito (libertad 0)
- La libertad de estudiar cómo funciona el programa, y adaptarlo a nuestras necesidades (libertad 1)[5]
- La libertad de redistribuir copias de manera de poder ayudar a nuestros vecinos (libertad 2)
- La libertad de mejorar el programa y publicar esas mejoras al público, de manera que toda la comunidad se beneficie (libertad 3)[2]

Para garantizar estas libertades la FSF crea un conjunto de licencias libres, entre las que se destacan la GPL (General Public License) y LGPL (Lesser GPL) [5] [7]

---

3 **FLOSS:** http://es.wikipedia.org/wiki/FOSS_-_FLOSS

4 **FSF**: http://fsf.org

5 El acceso al código fuente es una precondición para que se cumpla esta libertad

El movimiento Open Source (OSS) es filosóficamente distinto al movimiento del software libre. Comenzó en 1998 con un grupo de personas que formó la organización Open Source Initiative (OSI)[6]. La principal diferencia con el movimiento de Software libre es que éste último hace hincapié en aspectos morales y éticos del software, mientras que el movimiento Open Source afirma que la excelencia técnica del software se basa en poder disponer del código fuente. Considera el desarrollo Open Source como

> *" ... un método de desarrollo de software que aprovecha el poder de revisión de pares que están distribuídos y la transparencia del proceso. La promesa del open source es mayor calidad, más alta confiabilidad, más flexibilidad, menores costos, y el final de la dependencia de proveedores predadores."* [7]

En este trabajo, sin embargo, importan más las coincidencias que las diferencias existentes entre ambos movimientos, y por lo tanto a través de esta tesis usaremos el concepto FLOSS para referirnos indistintamente a software libre u open source.

El modelo de desarrollo FLOSS es utilizado para la construcción de aplicaciones para usuarios finales, servidores web, sistemas operativos, juegos, aplicaciones científicas, herramientas de comunicación, lenguajes de programación, librerías, entre otros, los cuales pueden ser usados, adaptados y extendidos libremente.

Estos principios de solidaridad guardan cierta similitud a los principios que dieron origen a Internet como un lugar abierto donde el conocimiento pueda circular libremente.

La creación e innovación en proyectos FLOSS puede ser mayor que en el modelo de desarrollo tradicional, ya que mayor cantidad de gente está adaptando el producto a sus necesidades, debido a la especial relación existente entre los desarrolladores y los usuarios que forman la comunidad, y principalmente por la posibilidad de reutilizar todo o partes de otros productos FLOSS. El resultado usualmente se traduce en software de alta calidad, aunque el principal beneficio es prevenir el surgimiento de productos monopólicos y asegurar el libre acceso al conocimiento tecnológico.

1.3. FLOSS como modelo de desarrollo

FLOSS, además de ser analizado desde el punto de vista de las licencias que garantizan ciertas libertades a los usuarios, también puede ser analizado como un innovador modelo de desarrollo [7]

Esta concepción ha sido estudiada por Eric Raymond en su ensayo "la catedral y

6 **OSI**: http://www.opensource.org/

7 Extraído y traducido de http://opensource.org

el bazar"[2], donde introduce el concepto de desarrollo compartido, en el que cada desarrollador es libre de elegir en que porción del código desea colaborar, al que denominó estilo bazar, y lo diferenció de otro estilo de desarrollo más formal, rígido y estructurado, al que denominó estilo catedral.

Raymond hace hincapié que en los proyectos desarrollados colaborativamente es posible compartir la realización de actividades principales tales como la codificación, y otras auxiliares como pueden ser el reporte y corrección de bugs, la documentación y traducción a diferentes idiomas, a través de una gran comunidad de usuarios/desarrolladores, posiblemente distribuídos. Continúa diciendo, los integrantes de estas comunidades especiales, usualmente lo hacen de manera voluntaria, proveen de su esfuerzo y recursos, y en el tiempo se van transformando en usuarios expertos con capacidad de realizar contribuciones importantes hacia su comunidad de origen.

Raymond sostiene que es el modo de desarrollo estilo bazar es el más adoptado por los proyectos basados en software libre, con una comunidad de usuarios usualmente descentralizada, que coopera en la construcción colaborativa del software, donde naturalmente surge una relación del tipo "ganar-ganar". Esta definición de modelo de relación "ganar-ganar" entre las personas que participan dentro de una comunidad de software libre, donde la participación y el grado de contribución son voluntarios, se sustenta en que todos y cada uno de los sus miembros pueden cosechar beneficios que se puedan medir, derivados de su participación en la comunidad.

Y la participación en una comunidad de software libre, no requiere más que conocer y estar de acuerdo con sus principios elementales. Estos son:

- Los proyectos de software libre públicamente accesibles a través de internet adoptan licencias de software bien conocidas y altamente contrastables, que garantizan la no apropiación del producto por un grupo de personas. Esta garantía de la mayoría, por intermedio de una licencia libre de software, es la que le asegura a la minoría, es decir al individuo o grupo de colaboradores interesados en participar del proyecto, que el producto total continuará existiendo en los mismos términos de la licencia original, y que no podrá ser "privatizado" por ninguna persona u organización.
- Los proyectos de desarrollo de software libre contemplan el reconocimiento público de la propiedad intelectual sobre el trabajo realizado por cualquier contribuyente al proyecto.
- Los proyectos de software libre motorizados por una comunidad de usuarios se construyen "en público", usando herramientas colaborativas en internet para coordinar actividades, intercambiar ideas, consultas, sugerir mejoras, informar y/o solucionar errores, acceder al código fuente del proyecto, a su documentación.

1.4. Comunidades de innovación

Según Eric von Hippel [3] las comunidades de innovación tienen como principal función facilitar el acceso a la información que los innovadores desean revelar.

Sintéticamente las define como *"nodos que consisten en individuos u organizaciones interconectadas a partir de transferir información que puede involucrar comunicaciones cara-a-cara, electrónicas o de otro tipo"*.

Estas comunidades pueden tener usuarios y/o proveedores como miembros y contribuyentes, y pueden surgir o florecer cuando alguien innova voluntariamente, revela sus innovaciones y cuando otros encuentran interesante tal información revelada. Tienen como característica la de ser usualmente especializadas, centralizando la información relacionada con las categorías de la innovación. Pueden consistir no sólo en repositorios de información, sino también ofrecer servicios adicionales tales como salas de chat y listas de correo donde los participantes puedan intercambiar ideas y proveerse de ayuda mutua. Normalmente se les provee a los miembros de la comunidad de herramientas de ayuda para desarrollar, evaluar e integrar sus trabajos.

Estas características generales de las comunidades de innovación son más visibles y notorias en las comunidades alrededor de proyectos FLOSS, y son éstas las que este trabajo pretende analizar.

1.5. Comunidades FLOSS

1.5.1. Herramientas colaborativas

Las **comunidades** alrededor de proyectos FLOSS **utilizan herramientas colaborativas** que ayudan en la tarea de organizar el trabajo cooperativo y descentralizado de multitud de personas.

Los proyectos más difundidos de software libre tienen una web que sirve de punto de encuentro para sus usuarios, desarrolladores o simplemente interesados, donde es posible encontrar los recursos necesarios para utilizar, conocer o intercambiar ideas con otros usuarios.Otros utilizan la infraestructura provista por servicios tales como SourceForge[8],Google Hosting[9], BerliOS[10], Savannah[11], entre otros. Cada proyecto suele tener listas de discusión, foros, canales de chat donde se discuten nuevas ideas, las mejoras, o simplemente donde los usuarios pueden realizar consultas. También ofrecen herramientas colaborativas donde el

8 **SourceForge**: http://sourceforge.net

9 **Google Hosting**: http://code.google.com/hosting

10 **BerliOS**: http://berlios.de

11 **Savannah**: http://savannah.gnu.org

conocimiento generado puede ser organizado y categorizado.

Es así que todo proyecto de software libre público utiliza algún gestor de versiones del software que facilita el trabajo concurrente de múltiples desarrolladores, poseen también una herramienta que permita realizar un seguimiento de los errores detectados en el software, y generalmente este tipo de proyectos tienden a crear la documentación con manuales de uso, guías, tutoriales, buenas prácticas, preguntas frecuentes, a través de un sistema web que permite la edición colaborativa de contenido abierto a partir de herramientas conocidas como WIKIs.

1.5.2. Conocimiento generado en los proyectos FLOSS

Estas comunidades “alrededor” de proyectos FLOSS, donde un grupo de desarrolladores y usuarios trabajan geográficamente dispersos, son apoyados por una red de interacciones. Los participantes generalmente son especialistas en el uso de tecnologías de información y comunicación, y construyen el software mediante producción entre pares y uso intensivo de herramientas colaborativas accesibles a través de una red interna (intranet) o externa (extranet, por ejemplo: internet).

En la construcción del software se genera conocimiento estructurado y no estructurado en la forma de documentos, guías, tutoriales, buenas prácticas y librerías de software que son almacenadas de diferentes maneras, con distintos formatos y en múltiples repositorios. Este **conocimiento** es un recurso fundamental para los miembros y desarrolladores de la comunidad[3][4]

1.5.3. Organización del conocimiento en FLOSS

El término "**conocimiento**" expresa cierta ambigüedad al consultar diferente bibliografía especializada, y en general coinciden en que significa mucho más que "lo que las personas saben o conocen" [4][6].

Cuando el término "conocimiento" es combinado con el término "administración" implica que las personas desean cuantificar, medir u organizar el conocimiento de manera que sea visto como un **objeto** que puede ser capturado y almacenado. Desde esta perspectiva el conocimiento es catalogado solamente como información. Este punto de vista del conocimiento como **objeto** está centrado en la tecnología, cuyo principal rol es la creación de repositorios de conocimiento estructurado.

Aunque existe otra visión que considera aspectos del conocimiento que no pueden ser capturados, codificados y almacenados, Kimble[4] argumenta que la visión del conocimiento como **objeto** continúa siendo dominante en el campo de administración del conocimiento.

Existe otro tipo de conocimiento denominado *conocimiento tácito,* el cual es informal, no estructurado, reside "en las personas" y es difícil, sino imposible, de reflejar formalmente en soporte digital.

La incorporación de herramientas colaborativas facilita el almacenamiento y estructuración del conocimiento surgido dentro de la comunidad del proyecto. Su uso facilita el diseño, administración, documentación y creación de una base de conocimientos que ayuda al desarrollo, *debugging*, seguimiento y *testing* de los proyectos FLOSS.

En los proyectos FLOSS que se desarrollan "virtualmente", los programadores, arquitectos, analistas, traductores, documentadores y diseñadores geográficamente distantes, coordinan esfuerzos y cooperan utilizando herramientas colaborativas sincrónicas y asincrónicas. En general, las herramientas colaborativas sincrónicas (Instant Messaging o chat por ejemplo), no tienen como característica principal almacenar en un medio concreto y no volátil la información relevante generada durante la interacción entre los participantes. No es el caso de las herramientas colaborativas asincrónicas (WIKI, Bug Tracker, Email, Foros, SCM) donde el intercambio de información, ideas, discusiones, quedan almacenadas en un medio durable, estructurado o no, de fácil recuperación posterior.

Sin embargo la mera inclusión de herramientas asincrónicas en proyectos de software no conlleva a una mejora, ayuda o facilitamiento del desarrollo de software.

Frecuentemente la información relevante para la toma de decisiones o solución de errores está **dispersa** en múltiples fuentes de información, y su búsqueda y recuperación no es trivial.

1.5.4. Organización dispersa del conocimiento

La necesidad que la información esté permanentemente disponible y accesible, presenta nuevos desafíos al desarrollo de aplicaciones. Las herramientas colaborativas asincrónicas utilizadas aisladamente no llegan a cubrir las nuevas necesidades que el crecimiento vertiginoso de los proyectos FLOSS están requiriendo.

La integración de aplicaciones no es una tarea sencilla, y presenta numerosos inconvenientes que se irán analizando conforme se avance en el desarrollo del tema del trabajo.

Existen numerosas estrategias, técnicas, patrones y tecnologías que han sido desarrollados durante años para afrontar los diferentes escenarios que pueden surgir en situaciones de integración de aplicaciones. Estas estrategias varían desde integración punto-a-punto, *hub&spoke* o basadas en *bus*, y tecnologías, patrones de diseño y estándares que incluyen EAI, *Web Services*, WS-*, SOA, ESB, JBI, entre otros.

Este trabajo al analizar las principales tecnologías y estrategias existentes de integración de aplicaciones, **enfocará en SOA, ESB y JBI** como las tecnologías más adecuadas para resolver el problema de integración e interoperabilidad entre herramientas colaborativas asincrónicas y heterogéneas.

La solución planteará la **necesidad y ventajas** en el uso de estándares abiertos y herramientas FLOSS para la implementación de una solución basada en ESB que permitirá construir una infraestructura de integración adaptable, flexible y escalable en el tiempo.

Para las implementaciones que acompañan este trabajo se utilizaron tecnologías estándares y abiertas y herramientas/frameworks basados en Java, y con licencias libres u open source.

1.6. Caso de Estudio implementado

1.6.1. Introducción

A fin de probar la funcionalidad provista por la versión preliminar de la plataforma de integración basada en un ESB estándar y tecnologías con licencias libres, se planteó un caso de estudio para un hipotético escenario donde se requiera implementar una funcionalidad de búsqueda federalizada, en base a tres posibles ámbitos de utilización.

1.6.2. Motivación del caso de estudio

En el área de la ingeniería de software algunos investigadores concuerdan en afirmar la inexistencia o dificultad de acceso a datos empíricos de proyectos de software. En el caso de proyectos FLOSS, con comunidades de usuarios y desarrolladores contribuyendo a su sostenimiento y crecimiento en el tiempo, la obtención de datos que puedan servir para investigaciones referidas a proyectos de software se simplifica al estar prácticamente toda la información disponible para su visualización. Sin embargo, la dificultad encontrada para el acceso y recuperación de tales datos radica en la **heterogeneidad** del formato de datos y herramientas utilizadas para almacenar toda la información asociada al proyecto FLOSS.

El caso de estudio presentado en la segunda parte de este trabajo pretende ser un prototipo de una plataforma de integración estándar y *open source* que permita realizar búsquedas federalizadas en las diversas herramientas colaborativas típicas de los proyectos FLOSS, y que sea lo suficientemente flexible, configurable y no restrictiva en cuanto a las tecnologías utilizables para realizar la búsqueda.

Este caso de estudio plantea tres escenarios de búsquedas posibles, para ilustrar posibles ámbitos de utilización. Estos escenarios serán analizados con mayor profundidad y detalle en la segunda parte de este informe, pero brevemente haremos en esta parte una introducción a su funcionalidad.

1.6.3. Escenarios del caso de estudio

Los tres escenarios de búsquedas planteados en el caso de estudio son: búsqueda

simple o basada en texto no estructurado, búsqueda basada en XML (texto estructurado pero con formato no estándar) y búsqueda basada en SOAP (texto estructurado y con formato estándar).

El escenario de búsqueda simple está pensado para ser usado por usuarios quienes sin conocer aspectos internos de la herramienta de búsqueda simplemente ingresan un texto o parte de él, seleccionan algunos parámetros adicionales (orden de los resultados, paginado, fuentes de información) y realizan la búsqueda. La respuesta obtenida es transformada a formato HTML mediante un servicio de transformación basado en plantillas XSL.

El escenario de búsqueda basado en documentos XML está pensado para procesos de software que requieran realizar la búsqueda enviando la consulta a través de un documento XML con un formato acordado previamente. Este formato no es estándar, sino que es definido por quien despliegue el prototipo de la plataforma de integración, y particularmente de la implementación del servicio de búsqueda que procesa documentos XML.

El escenario de búsqueda basado en mensajes SOAP es similar a la búsqueda basada en XML, pero el cliente debe enviar mensajes SOAP construidos en base a un documento WSDL publicado por la plataforma de integración.

Los escenarios de búsqueda basados en XML y SOAP, en el caso de estudio implementado, transforman el formato de sus resultados (XML y SOAP respectivamente) a HTML, mediante servicios de transformación basados en XSL, ya que de esta manera estos escenarios pueden ser fácilmente accedidos y probados mediante un cliente web. En un escenario real, ambas búsquedas retornarían un documento XML y SOAP respectivamente, y sería el cliente PC el encargado de analizar y mostrar el resultado de manera apropiada.

1.7. Mención a este trabajo

Este trabajo fue seleccionado para participar en la edición 2008 del concurso iberoamericano de innovación y tecnologías sustentables organizado por ISTEC[12].

La final latinoamericana se realizó en Guayaquil, Ecuador, donde este trabajo fue seleccionado para participar de la posterior etapa final iberoamericana realizada en Cancún, México, en el mes de Diciembre de 2008.

Por este trabajo obtuve una mención especial y un premio otorgado por SUN Microsystems[13]

12 **ISTEC**: http://istec.org

13 http://www.istec.org/events/ga/conibes/home

Capítulo 2

Integración de Aplicaciones

2.1 Introducción

La **integración de aplicaciones** surge ante la necesidad de conectar múltiples **sistemas**, generalmente **heterogéneos**, permitiendo que éstos cooperen y provean funcionalidad unificada. Existen numerosos factores que hacen que esta tarea sea sumamente dificultosa y compleja: las aplicaciones pueden estar implementadas enlenguajes de programación diferentes; pueden estar geográficamente distantes; pueden ejecutarse sobre diferentes ambientes de ejecución (software y/o hardware); pueden utilizar formato de datos no compatible entre sí; pueden no tener documentación o estar desactualizada generalmente por ser aplicaciones heredadas [8][9][14][37].

Usualmente otro factor que hace aún más dificultoso el proceso de integración es la imposibilidad de introducir cambios en las aplicaciones para que puedan incorporarse a una infraestructura de integración. Las razones de esta limitación está relacionada con la complejidad de la lógica de negocio de determinados sistemas, o del manejo de datos sensibles que estos sistemas hacen.

El presente capítulo intenta brindar una breve introducción sobre los conceptos relacionados con la integración de aplicaciones. Esta descripción permitirá al lector tener una visión general de los temas tratados y obtener el conocimiento necesario sobre algunos de los principales conceptos y terminología utilizada a lo largo de este trabajo.

Algunos de los conceptos nombrados en este capítulo serán abordados con mayor profundidad en capítulos posteriores, especialmente **SOA** y **ESB**, centrales en el desarrollo de esta tesis.

2.2. Débil Acoplamiento

Débil acoplamiento es un término actualmente muy relacionado con el área de integración de aplicaciones. La idea detrás de este término, en relación a la integración de aplicaciones, es que las aplicaciones que necesiten comunicarse o interactuar entre sí minimicen o directamente no asuman nada acerca de la o las aplicaciones con las cuales necesita comunicarse.

En un ambiente débilmente acoplado, las aplicaciones pueden ser modificadas independientemente sin que ésto signifique hacer cambios en las demás aplicaciones. Sin embargo, el débil acoplamiento refiere también a otros aspectos en la relación entre las aplicaciones: la integración de aplicaciones también debe ser débilmente acoplada en cuanto a los tiempos de ejecución, no requiriendo que la disponibilidad de las aplicaciones cooperantes en el mismo instante de tiempo; tampoco deben existir restricciones en cuanto al formato de datos de cada una de ellas, cada aplicación debería poder utilizar su modelo de datos sin necesidad de adaptarlo para poder integrarse con otras aplicaciones.

El concepto opuesto a débil acoplamiento se conoce como ***fuerte acoplamiento***. En este caso las aplicaciones realizan múltiples asunciones respecto a las demás y a los protocolos de comunicación utilizados; es así que las comunicaciones serán más eficientes, pero, en general, la solución será menos tolerante a interrupciones o cambios en las aplicaciones participantes.

Muchas tecnologías de integración que serán analizadas posteriormente, utilizan la misma semántica, fuertemente acoplada, de invocación a métodos locales, para realizar invocaciones remotas para el intercambio de datos entre aplicaciones/servicios. Surgen las nociones de RPC (*Remote Procedure Call*), RMI (*Remote Method Invocation*), soportadas por distintos frameworks y plataformas, tales como CORBA, Microsoft DCOM, .NET Remoting, Java RMI, entre otros. La principal ventaja de esta estrategia de integración es lo natural que le resulta a los desarrolladores comenzar a utilizar estas tecnologías, al tener similitudes con invocaciones locales a métodos.

Sin embargo deben considerarse algunas cuestiones importantes relacionadas con este modelo de integración puramente sincrónico. Por ejemplo, no es deseable que las aplicaciones deban esperar hasta que los resultados de la invocación remota estén disponibles. El tiempo de espera puede no ser aceptable para determinados escenarios, e incluso, en el caso de interrupciones en la comunicación, la aplicación puede quedar suspendida esperando por una respuesta.

Otra asunción realizada en el caso de acoplamiento fuerte es que la aplicación que invoca el método remoto, como la que implementa el método invocado, deben estar codificadas con el mismo lenguaje de programación, limitando de esta manera los escenarios de integración posibles.

De esta manera el débil acoplamiento es un concepto esencial en toda estrategia de integración de aplicaciones.

Sin embargo, a pesar de las evidentes ventajas de esta “buena práctica” (flexibilidad, escalabilidad, reusabilidad, tolerancia a fallos, entre otras), el diseño de una estrategia de integración débilmente acoplada es una tarea sumamente dificultosa, que se traduce en *debugging* y *testing* más fuertes e intensivos, y un desarrollo más complejo.

En el capítulo sobre SOA se profundizará sobre algunos de los beneficios de las arquitecturas débilmente acopladas.

2.3. Integración: Evolución

En los años 90's con la maduración de las redes de comunicación y el surgimiento de estándares de interoperabilidad, se hizo popular el concepto de integración de aplicaciones. Fueron creadas metodologías, buenas prácticas y estándares para que la integración de sistemas sea posible y de manera eficiente.

A principios del 2000 comienza la era de Integración de Aplicaciones Empresariales (EAI), que permitió reemplazar la integración punto-a-punto por una estrategia más flexible y centralizada usando el concepto de *hub&spoke*. Sin embargo los protocolos de comunicación y *brokers* de integración continuaban siendo privativos.

El advenimiento de XML y la generación de buenas prácticas de interoperabilidad entre Web Services (WS-I)[14] permitió la estandarización de componentes para asegurar la independencia de las plataformas.

Actualmente las herramientas y estándares de integración giran alrededor de tecnologías de *middleware* distribuidas como ESB (*Enterprise Service Bus*), que enmarcada dentro de una SOA (*Service oriented Architecture*) brindan una solución de integración flexible y escalable en el tiempo [37].

2.4. Estrategias de Integración

La introducción realizada hasta el momento describe los principales aspectos de la integración de aplicaciones, junto a distintos factores a tener en cuenta al momento de plantear una estrategia de integración [35][36].

En esta sección se describirán los diferentes estilos de integración que podrán adoptarse para hacer frente al desafío de integrar aplicaciones heterogéneas.

Pero antes de seleccionar el estilo que mejor se adecúe a los requerimientos de la problemática, deben evaluarse algunos criterios importantes, y principalmente si, según Gregor Hohpe[8], es realmente necesaria una estrategia de integración. Si se requiere el desarrollo de una única aplicación, *standalone,* que no necesita

14 **WS-I**: http://ws-i.org

colaborar con otras aplicaciones, entonces, probablemente no sea necesaria plantearse una estrategia de integración. Sin embargo, la realidad indica que hasta las más simples organizaciones utilizan múltiples y heterogéneas aplicaciones, que necesitan cooperar para en determinados momentos mostrar información unificada, ya sea a usuarios, clientes, socios u otros.

Los que siguen son algunos criterios que deben tenerse en cuenta al momento de elegir una estartegia de integración:

- Acoplamiento entre aplicaciones
- Simplicidad de la solución de integración
- Tecnologías de integración
- Formato de datos
- Tiempo de entrega de datos
- Datos o Funcionalidad
- Asincronismo

2.4.1. Acoplamiento entre aplicaciones

Este tópico ya fue introducido anteriormente al describirse el concepto sobre acoplamiento débil . Básicamente este criterio establece que la principal regla a tener en cuenta al momento de seleccionar el estilo de integración a utilizarse, es que debe obtenerse el mayor grado posible de desacoplamiento entre las aplicaciones que formarán parte de la solución de integración. Ésto asegurará cierto grado de flexibilidad, que permitirá que cambios realizados dentro de una aplicación no afecten a la solución de integración global, ni a los mecanismos de comunicación e intercambio de datos entre las aplicaciones.

2.4.2. Simplicidad de la solución de integración

Cuando se decide sobre el estilo de integración a utilizar, la simplicidad de la solución de integración es un factor que debe considerarse. No sólo es definida por la complejidad del diseño y tecnologías seleccionadas, sino también por el número de cambios que deben realizarse en las aplicaciones integradas en la solución.

2.4.3. Tecnologías de integración

Las tecnologías que aplican al área de integración de aplicaciones están en constante evolución y cambio. Nuevos productos y estándares son introducidos constantemente.

Luego, es necesario evaluar los riesgos de incorporar nuevas tecnologías, que requieren un proceso de aprendizaje para obtener experiencia, e incluso pueden no estar lo suficientemente probadas para ser usadas en ambientes productivos.

2.4.4. Formato de datos

Las aplicaciones que participan en la solución de integración necesitan intercambiar datos para interactuar, los cuales deben ser "entendibles" por todos los participantes. De esta manera, es necesario un formato de datos uniforme y entendible por todas las aplicaciones.

Este requerimiento puede concretarse a través del uso de un formato de datos acordado entre todas las aplicaciones, o un componente que traduzca los formatos específicos de las aplicaciones.

La primera opción es la más intrusiva, ya que requiere cambios internos en las aplicaciones, lo cual atenta contra el principio de simplicidad descripto anteriormente.

2.4.5. Tiempo de entrega de datos

Un criterio importante a tener en cuenta en la construcción de la solución de integración es el tiempo. Particularmente el período de tiempo desde el momento que los datos son publicados hasta el momento que esos datos son consumidos por otra/s aplicación/es.

Este factor es importante para la *performance* general de la solución de integración. Mientras más corto es este período de tiempo, más rápidamente las aplicaciones recibirán los datos, los procesarán y retornarán los resultados.

2.4.6. Datos o Funcionalidad

Otro criterio a tener en cuenta es si las aplicaciones compartirán datos o funcionalidad. Compartir datos es una tarea más sencilla que compartir funcionalidad. La diferencia entre ambas técnicas serán discutidas en la sección siguiente, cuando se analicen los estilos de integración.

2.4.7. Asincronismo

Este criterio debe evaluarse para la elección de la forma en que las aplicaciones se comunicarán. Existen dos posibilidades:

- Comunicación Sincrónica
- Comunicación Asincrónica

En una comunicación sincrónica una aplicación invoca funcionalidad provista por otra aplicación remota, y espera hasta que procese los resultados y los retorne. Luego de recibir los resultados la aplicación "remitente" continua su procesamiento normalmente.

En una comunicación asincrónica la aplicación continua con sus actividades luego de enviar el pedido, sin necesidad de esperar por los resultados generados por la aplicación remota. Cuando los resultados están disponibles la aplicación "remitente" es notificada por algún medio.

Fire&Forget es un patrón de diseño en donde la comunicación es asincrónica, y la aplicación "remitente" no se interesa por los resultados retornados.

2.5. Estilos de Integración

Los **estilos de integración** propuestos por Martin Fowler[15] son solamente cuatro, y varían en cuanto al nivel de complejidad del diseño e implementación de la solución de integración. Los criterios presentados anteriormente pueden ser útiles al momento de seleccionar uno o varios de los estilos que se analizarán a continuación [26]

Las aproximaciones al problema de integración pueden dividirse en:

- Transferencia de archivos
- Base de datos compartidas
- Invocación a procedimientos remotos (RPC)
- Mensajería

2.5.1. Transferencia de archivos

Una aplicación lee el archivo escrito por otra aplicación. Es necesario que las aplicaciones coordinen el formato del contenido del archivo, el nombre, ubicación, el momento de escritura y lectura (para evitar colisiones) y el responsable de eliminar el archivo una vez utilizado.

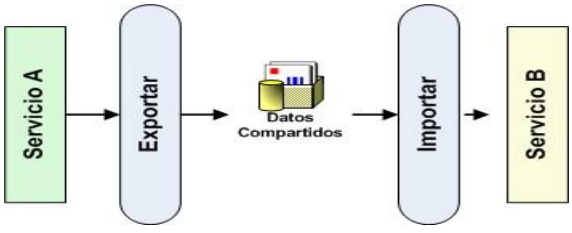


FIGURA 2.1 - Transferencia de archivos (adaptada de [87])

15 **Martin Fowler**: http://martinfowler.com

2.5.2. Base de datos compartidas

En esta estrategia las aplicaciones involucradas interactúan a través del uso compartido de un esquema de base de datos.

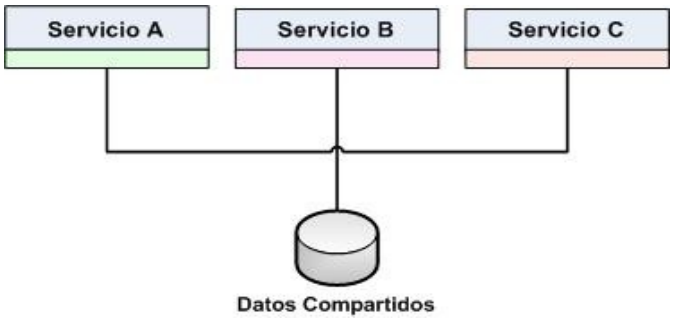


FIGURA 2.2 - Base de datos compartidas (adaptada de [87])

2.5.3. Invocaciones a procedimientos remotos

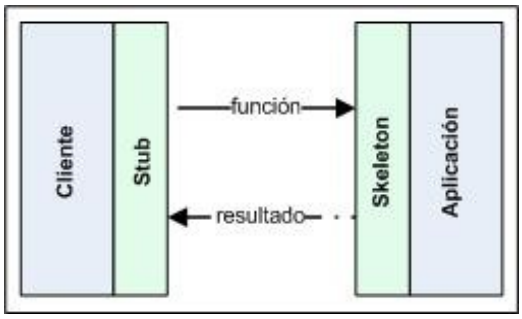


FIGURA 2.3 - Invocación remota a procedimientos (adaptada de [87])

En este estilo de programación un objeto y sus métodos (o procedimiento y parámetros) se consideran "remotos" si pueden ser invocados a través de una red.

Una aplicación utiliza un *proxy* local que imita la interface del objeto remoto y sus métodos. La aplicación que realiza la invocación es conocida como "cliente", y la implementación remota denominada "servicio" o "servidor".

El cliente realiza la invocación, de igual manera que una invocación local, que es serializada a través de la red al método o procedimiento remoto. De la misma manera los valores de retorno son serializados y enviados al cliente.

Este estilo de programación tiene la ventaja que los procedimientos remotos son publicados de tal manera que imitan la arquitectura de objetos subyacente de la aplicación involucrada, permitiendo al desarrollador realizar invocaciones a métodos en el lenguaje nativo.

Las invocaciones sincrónicas distribuidas RPC permiten obtener una respuesta de manera inmediata, indicando si la operación tuvo éxito o no. Sin embargo, en un ambiente distribuído existen escenarios de falla que en ambientes no distribuídos no suceden. Cuando se realiza una invocación sincrónica entre múltiples procesos, el éxito de una invocación RPC puede ser que dependa del éxito de otras invocaciones RPC que forman parte del ciclo *Request/Response*. Este ciclo se cumple si todas las invocaciones tuvieron éxito, y falla, si alguna no pudo completarse.

RPC aplica el principio de encapsulamiento para la integración de aplicaciones. Si una aplicación necesita información almacenada en otra aplicación, la pide directamente a esa aplicación. Si una aplicación necesita modificar información almacenada en otra aplicación, lo hace a través de una invocación a dicha aplicación. Cada aplicación es responsable de mantener la integridad de su información.

Incluso, cada aplicación puede alterar su información sin que se vean afectadas las demás aplicaciones.

Existen diferentes alternativas de implementaciones RPC. Entre las más conocidas figuran CORBA, COM, Java RMI. Los servicios web a través de SOAP también soportan interacciones del tipo RPC.

Sin embargo, aunque el principio de encapsulamiento ayuda a reducir el acoplamiento entre aplicaciones al eliminar grandes estructuras de datos compartidas, todavía sigue existiendo acoplamiento debido a las invocaciones remotas de las funciones expuestas por las aplicaciones.

En un ambiente RPC fuertemente acoplado cada aplicación debe conocer detalles internos de las demás aplicaciones con las que desee comunicarse, por ejemplo, la cantidad de métodos que publica y detalles de los parámetros que cada método acepta.

A medida que crece el número de aplicaciones y servicios a integrar, también crece el número de interfaces que necesitan ser creadas y mantenidas.

Cada interface refiere a los métodos que una aplicación hace públicos para ser invocados. Si cada objeto publica un solo método en su interface, entonces el número de puntos de conexión coincidiría con el número de interfaces. Este número sería la cota inferior del número de puntos de conexiones posibles.

2.5.4. Mensajería

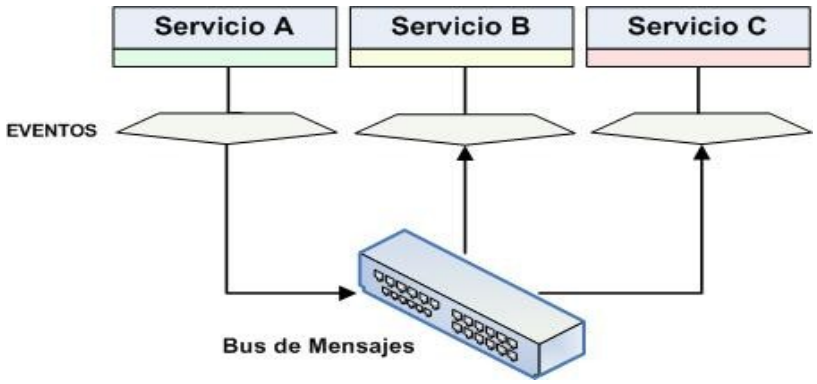


FIGURA 2.4 - Estilo de integración mediante el paso de mensajes (adaptada de [87])

La mensajería es considerada el estilo de integración que realiza la menor cantidad de asunciones respecto a los demás participantes. Además de este hecho, es la técnica actualmente más sofisticada que puede utilizarse para la resolución al problema de integración.

La mensajería ofrece numerosas ventajas respecto a los estilos de integración antes mencionados. En particular las bases de datos compartidas acoplan las aplicaciones a una base de datos, mientras que RPC asume muchas cosas respecto a las partes involucradas en la interacción.

Por el contrario, la mensajería esta librada de estos inconvenientes; ofrece una manera eficiente y fiable de comunicar aplicaciones, en donde se puede transmitir una gran cantidad de datos separados en pequeñas unidades, siendo el receptor notificado si existen más datos por procesar, o es posible reintentar el envío de datos para asegurar que lleguen correctamente.

El modelo de comunicación que utiliza es asincrónico, por lo tanto la aplicación "remitente" no se bloquea a la espera que el receptor reciba, procese y retorne los resultados; tampoco requiere que ambas aplicaciones estén disponibles al momento de realizar la interacción. En definitiva, la mensajería hace uso de todos los beneficios y características nombradas en la sección 2.2 sobre **débil acoplamiento**, y en la sección 2.4.7 sobre **asincronismo.**

2.6. Sistemas basados en Mensajería

El sistema de mensajería es el encargado de transportar y garantizar el envío y entrega de los mensajes, lo cual supone una responsabilidad menos para las aplicaciones intervinientes en la estrategia de integración al no tener que

preocuparse en esta tarea. [26][28][29][36][44]

2.6.1. Conceptos básicos de Mensajería

- **A. Canales de mensajería**: las aplicaciones de mensajería transmiten los datos a través de canales de mensajería, que permiten conectar el remitente con el receptor.

- **B. Mensajes**: un mensaje es un paquete atómico de datos transmitidos a través del canal. Para transmitir datos, una aplicación debe dividir los datos en uno o más paquetes, envolver cada paquete como un mensaje, y luego enviar ese mensaje por el canal apropiado. La aplicación receptora extrae los datos del mensaje y los procesa. El sistema de mensajería se encarga de la entrega del mensaje, aún en casos de fallas.

- **C. Entrega en múltiples pasos**: en determinadas ocasiones, los mensajes necesitan de ciertas acciones luego de ser enviados por el remitente original y antes de ser entregados al receptor final. Una arquitectura de tuberías (*pipes*) y filtros (*filters*) describe como múltiples pasos pueden ser encadenados usando canales.

- **D. Ruteo de mensajes**: cuando existen numerosas aplicaciones y canales, el mensaje puede atravesar varios canales hasta alcanzar su destino final. En estos casos el remitente puede no saber que canal utilizar para enviar los mensajes. Se hace uso de un componente intermedio, *Message Router*, que determina como navegar la topología de canales y garantizar que el mensaje llegue al destino final, o al router más próximo para continuar el camino.

- **E. Transformación de mensajes**: puede ocurrir que los formatos de datos de diferentes aplicaciones no sean compatibles. El remitente envía el mensaje en un formato en particular, mientras que el receptor espera un formato diferente. En estos casos el mensaje pasa por un filtro intermediario que convierte entre distintos formatos.

- **F*. Endpoints***: ubicación abstracta del servicio.

A. Canales de Mensajería

Cuando una aplicación quiere comunicarse y transmitir información a otra aplicación conectada al sistema de mensajería, agrega la información a un canal de mensajes en particular. La aplicación receptora lee los datos que espera recibir del mismo canal.

FIGURA 2.5 – Canal de mensajería entre una aplicación remitente y otra destino

El sistema de mensajería tiene diferentes canales de mensajes para distintos tipos de información que las aplicaciones desean comunicar. La aplicación que desea transmitir cierta información lo hace por el canal especifico para ese tipo de información, mientras que la aplicación receptora busca la información que espera recibir en el canal apropiado.

Los canales son direcciones lógicas dentro del sistema de mensajería, cuya implementación dependen del sistema de mensajería utilizado.

Decisiones al momento de usar los canales de mensajes

1. Tipos de canales

Existen 2 alternativas de canales de mensajes: canal punto-a-punto o canal publicador/subscriptor

El canal punto-a-punto no garantiza que el mensaje sea siempre recibido por el mismo receptor (puede haber más de 1 receptor leyendo del canal), pero si asegura que va a ser recibido solamente por una de las aplicaciones.

El canal publicador/subscriptor permite el envío de un mensaje que será copiado a todos los subscriptores (receptores) de ese canal

2. Tipos de datos

El contenido de los mensajes debe adherir a algún tipo de datos de manera que el receptor sepa cual es la estructura de los datos transmitidos.

Para que el receptor sepa como procesar el contenido de los mensajes que recibe, todos los datos transmitidos por un canal en particular deben ser del mismo tipo de datos.

3. Garantía de entrega de mensajes

Una de las principales ventajas de la mensajería asincrónica con respecto a RPC es que el remitente, receptor y red no deben estar funcionando todos al mismo tiempo al momento de la interacción. Si la red o el receptor no están disponibles al momento de realizar el envío, el sistema de mensajería debe almacenar el mensaje hasta tanto vuelvan a estar disponibles estos elementos.

Los mensajes no entregados son almacenados por el sistema de mensajería. Este proceso de denomina *store&forward*. La estrategia más simple de almacenamiento es utilizando memoria volátil. Esta estrategia no es confiable ya que el propio sistema de mensajería puede fallar.

La otra estrategia se denomina "*entrega confiable*". En este caso existe un almacenamiento de datos seguro en las computadoras donde se está ejecutando el sistema de mensajería. Cuando el remitente envía un mensaje la operación de envío no se considera completa hasta tanto el mensaje sea almacenado en el almacenamiento local del remitente. De la misma manera los mensajes no son borrados del almacenamiento hasta tanto los mensajes hayan sido entregados correctamente.

Aunque esta estrategia asegura la confiabilidad en la transmisión de mensajes, reduce la performance de las comunicaciones y consume espacio extra de disco.

4. Adaptadores de canales

Si una aplicación cliente no puede conectarse al sistema de mensajería, pero necesita hacerlo, es posible realizarlo a través de los adaptadores de canales. Requiere que el sistema de mensajería tenga alguna manera de conectarse con la aplicación, ya sea por medio de una conexión de red TCP/IP, HTTP, una base de datos, API u otros.

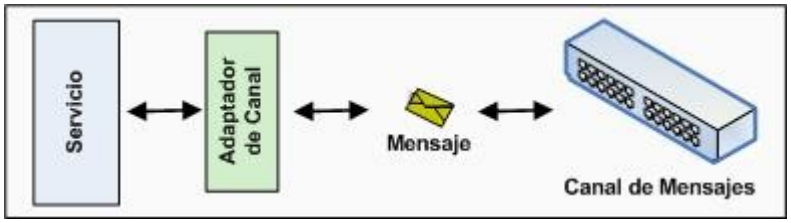


FIGURA 2.6 – Adaptador de canal entre la aplicación remitente y el sistema de mensajería

5. Bus de Mensajes

Un bus de mensajes es un componente de infraestructura del sistema de mensajería que facilita la comunicación entre aplicaciones diferentes a partir de un conjunto de interfaces compartidas.

Existen elementos necesarios y esenciales que facilitan la creación y utilización del bus:

- **Infraestructura de comunicación:** un sistema de mensajería típicamente es utilizado como una infraestructura de comunicación que sea transparente en cuanto a plataformas de hardware/software, lenguajes de programación y formatos de datos de las aplicaciones que integra. Puede incluir un router de mensajes que facilite el correcto enrutamiento de mensajes entre aplicaciones. El bus de mensajes es un componente esencial en este contexto al brindar el soporte requerido para las comunicaciones entre los demás componentes de infraestructura, y las aplicaciones que participen de la estrategia de integración.

- **Adaptadores**: las aplicaciones que quieran integrarse deben proveer de una interface que permita conectarlas al bus. Generalmente ésto es realizado a través de adaptadores que pueden ser propietarios o desarrollados a medida.

- **Comandos comunes:** son necesarios ciertos comandos comunes entendibles por todos los participantes conectados al bus, que permitan normalizar la comunicación entre las aplicaciones. Generalmente, la mejor manera de manejar estos comandos es a través de la creación de adaptadores para que las aplicaciones puedan comunicarse con el bus y viceversa.

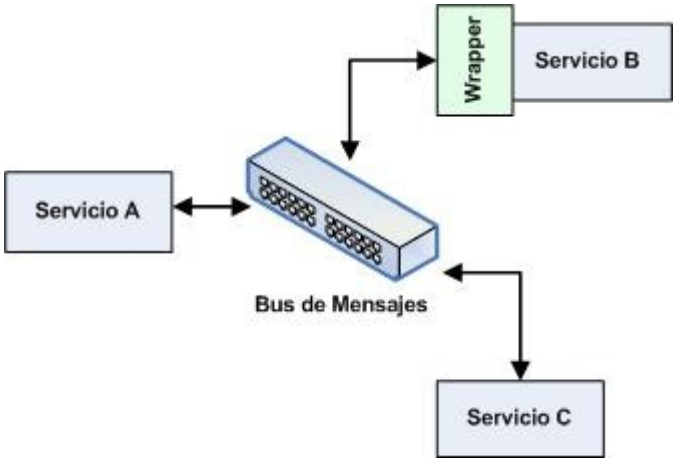


FIGURA 2.7 – Bus de mensajes que conecta aplicaciones y servicios diversos

B. Mensajes

Un canal de mensajes puede pensarse como un conducto que conecta 2 aplicaciones y permite que la información fluya entre ambos extremos.

En escenarios en donde se comparte el espacio de memoria *heap*, el pasaje de información es relativamente sencillo; tal es el caso del envío de parámetros a una invocación de función, o el compartir objetos entre *threads* de un mismo proceso.

Cuando las aplicaciones o procesos no comparten memoria, el intercambio de

información se vuelve complejo. Es necesaria la copia de información de un espacio de memoria a otro distinto.

Generalmente los datos son transmitidos como un *stream* de *bytes*, lo cual significa que el proceso remitente debe codificar los datos, y el proceso receptor debe decodificarlos a su formato original. El proceso de codificación es la manera en que las invocaciones RPC envían parámetros al procedimiento remoto, y reciben los resultados.

Al empaquetar los datos en uno o más mensajes es posible enviarlos a través de un canal de mensajes dentro de un sistema de mensajería.

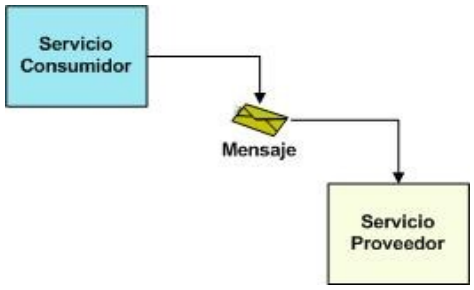


FIGURA 2.8 – Mensaje que circula por el sistema de mensajería

Un mensaje consiste de 2 partes básicas:

- **Encabezado**: contiene la información utilizada por el sistema de mensajería que describe los datos siendo transmitidos, el destino, origen, y toda metadata necesaria.
- **Cuerpo**: los datos que quieren ser transmitidos.

Para el sistema de mensajería todos los datos son iguales. Sin embargo, para las aplicaciones existen diferentes tipos de mensajes: *command message, document message, event message.*

B.1. Tipos de mensajes

- Estilo ***Command***: típicamente usado en escenarios donde es necesario invocar un procedimiento de otra aplicación. El patrón *Command*[16] especifica como encapsular un pedido en un objeto que puede ser almacenado o enviado. Si este objeto es un mensaje puede ser enviado a través de un canal de mensajes. [26]

16 **Patrón de diseño Command:** es.wikipedia.org/wiki/Command_(patrón_de_diseño)

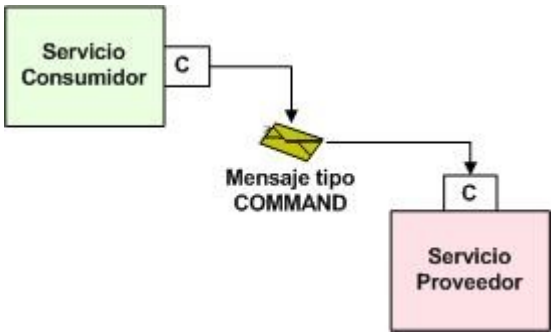


FIGURA 2.9 – Mensaje tipo *Command*

- Estilo ***Document***: el estilo de mensaje *Command* es intrínsecamente sincrónico, lo cual obliga al receptor a estar ejecutándose para que la interacción pueda llevarse a cabo. La mensajería utilizando mensajes estilo *Document* es más confiable. Solamente se limita a pasar datos dentro del mensaje, y deja que el receptor los procese como quiera.

- Estilo ***Event***: este tipo de mensajes es útil en situaciones en que una aplicación desea notificar a otras de algún cambio. El patrón *Observer*[17] especifica como diseñar una estrategia que ofrezca soluciones a este escenario. Aunque es posible utilizar RPC para realizar las notificaciones, requiere que el/los receptor/es acepten la notificación inmediatamente.

  La mejor alternativa es enviar las notificaciones asincrónicamente, como un mensaje. Cuando una aplicación tenga un evento que notificar, construye un objeto evento, lo encapsula dentro de un mensaje, y lo envía al canal apropiado. Los observadores de ese evento van a recibir el mensaje por el mismo canal, extraen el evento y lo procesan.

  Las principales diferencias entre un mensaje estilo *Event* y otro estilo *Document* es el contenido y el momento de envío. En los mensajes estilos *Event* el contenido en determinadas circunstancias no es importante, e incluso puede estar vacío; de esta manera se utilizan estos mensajes para señalizar la ocurrencia de algún evento. El momento de envío de estos mensajes *Event* es importante. Debe ser enviado al momento que el cambio asociado ocurra, y los receptores deberían procesarlo lo antes posible.

B.2. Consideraciones en la creación de los mensajes

- **Intención del mensaje**: esta consideración hace referencia a la intención que tiene el remitente del mensaje con respecto a que espera que haga el receptor cuando reciba el mensaje.

  Puede enviarse el mensaje del estilo *Command* especificando la función a invocar en el receptor. El remitente puede enviar un mensaje tipo *Document* que incluya estructuras de datos y pueda ser procesado asincrónicamente. O puede enviar un mensaje del estilo *Event* para que el

17 **Patrón de diseño Observer**: es.wikipedia.org/wiki/Observer_(patrón_de_diseño)

receptor sea notificado ante cambios en el remitente.

- **Retorno de respuesta**: cuando una aplicación realiza un pedido, típicamente espera una respuesta notificando la llegada del mensaje, y eventualmente, los resultados del procesamiento. Este es el típico escenario de interacción *Request-reply.*

  El pedido generalmente se realiza a través de un mensaje estilo *Command* y la respuesta contenida en un mensaje estilo *Document*, o bien conteniendo la excepción producida. El remitente puede especificar una dirección de retorno indicando el canal por el cual espera recibir los resultados, y en caso que sean múltiples los pedidos realizados y en proceso, la respuesta puede contener un ID de correlación que permita relacionar el pedido y la respuesta (útil en mensajería asincrónica)

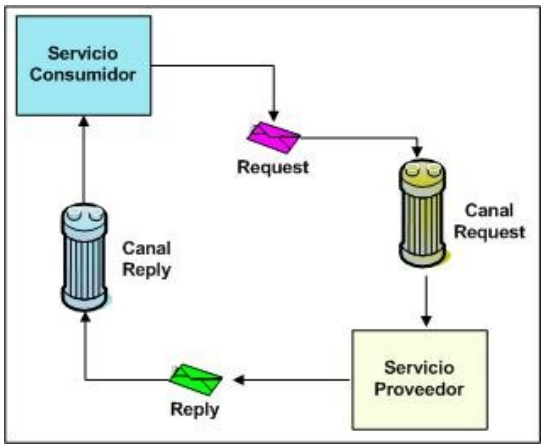


FIGURA 2.10 – Patrón Request/Reply entre aplicaciones

- **Cantidad de datos transmitidos**: en caso que el mensaje a transmitir contenga gran cantidad de datos, éstos pueden ser divididos en porciones más pequeñas y ser enviados como una secuencia de mensajes, de tal forma que puedan ser reconstruidos por el receptor.

  Los campos de identificación de un mensaje secuencia son:

  - **Identificador de Secuencia**: permite identificar la secuencia de otras secuencias de mensajes.
  - **Identificador de Posición**: dentro de la secuencia, identifica unívocamente y ordena cada parte
  - **Tamaño o indicador de fin**: especifica el número de mensajes que incluye la secuencia o marca el último mensaje.

C. Entrega en múltiples pasos

C.1. Tuberías y Filtros (*pipes* y *filters*)

El uso de tuberías y filtros permite realizar procesamiento en los mensajes antes que sean enviados a su destino final.

Cada tubería o filtro implementa una funcionalidad concreta y autónoma, para posibilitar su reuso en distintos escenarios.

Sin embargo, aunque la funcionalidad provista por estos componentes sea autónoma, aun pueden existir dependencias entre ellos en caso que el funcionamiento de algún componente requiera resultados provistos por otro componente.

Para evitar estas dependencias no deseadas, es necesario componer los componentes en una secuencia de pasos de procesamiento de tal manera que cada componente sea independiente de los demás, y los pasos puedan ser intercambiados sin afectar el resultado final.

La mensajería asincrónica ayuda a evitar dependencias entre componentes al no necesitar que los componentes queden a la espera de resultados provistos por otros componentes.

El uso de *pipes* y *filters* permite dividir una tarea de procesamiento grande, en pequeños pasos de procesamiento independiente (*filters*), conectados a través de canales (*pipes*).

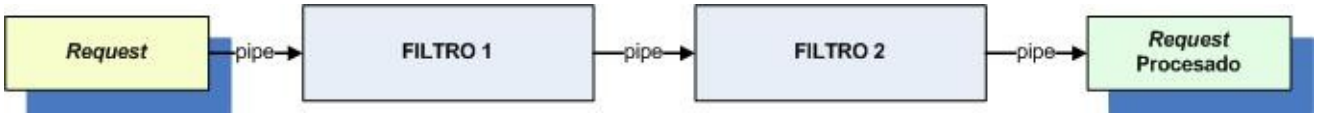


FIGURA 2.13 - Procesamiento basado en filtros y tuberías

Cada filtro expone una interface simple: recibe mensajes desde el canal entrante, procesa el mensaje y publica el resultado por el canal saliente. El canal se encarga de conectar los filtros.

Al usar interfaces estándares es posible combinar los componentes de manera flexible, mediante el agregado de nuevos filtros, omitiendo o reagrupando filtros existentes, sin necesidad de cambiarlos.

C.2. Procesamiento *Pipeline*

Conectar componentes a través de canales de mensajes asincrónicos permite realizar procesamiento concurrente de mensajes.

Cada componente puede procesar su mensaje, y al terminar, enviarlo al canal de salida e inmediatamente continuar procesando el siguiente mensaje, sin necesidad de esperar que el mensaje anterior termine de ser procesado por los demás componentes que forman parte de la secuencia de procesamiento.

Esta configuración llamada procesamiento *pipeline* (FIGURA 2.14) mejora notablemente la performance en relación al procesamiento secuencial estándar.

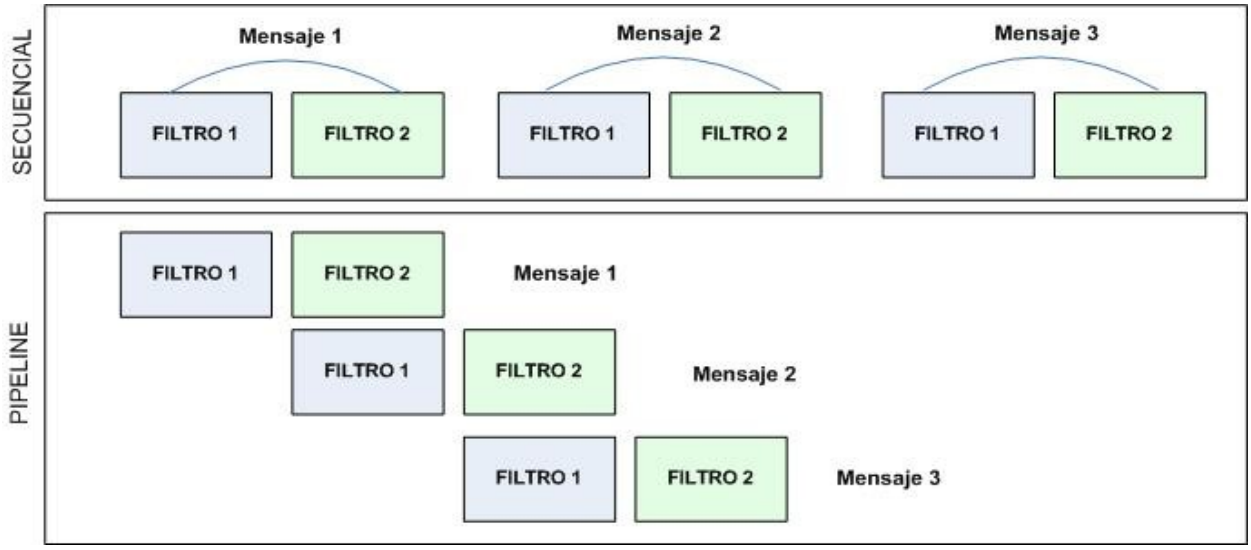


FIGURA 2.14 – Procesamiento *pipeline* vs Procesamiento secuencial

D. Router de mensajes

La arquitectura de *pipes y filters* conecta filtros directamente con otros a través de tuberías fijas.

Esta estrategia funciona bien para conjuntos de datos que siguen los mismos pasos de procesamiento secuencial. Sin embargo, en soluciones de integración basadas en mensajería, ésto no siempre es así. Los mensajes individuales pueden requerir diferentes series de pasos de procesamiento.

Un canal de mensaje permite desacoplar el remitente del receptor del mensaje. Esto significa que múltiples aplicaciones pueden publicar mensajes en un canal de mensajes.

Un canal de mensajes puede contener mensajes desde diferentes fuentes que pueden requerir distinto tratamiento en base al tipo de mensaje u algún otro criterio.

Una posible solución a este inconveniente es la creación de canales de mensajes para cada tipo de mensaje y conectar el canal a los pasos de procesamiento adecuado.

Sin embargo, esta alternativa requiere que los remitentes de los mensajes necesiten conocer el criterio de selección que cada pasos de procesamiento requiere, a fin de enviar el mensaje por el canal correcto.

La mejor alternativa es la creación de un filtro especial que consuma mensajes

por un canal de mensajes y lo republique a otro canal de mensajes dependiendo de un conjunto de condiciones. Este filtro especial, en ambientes de integración, se denomina **Router de Mensajes.**

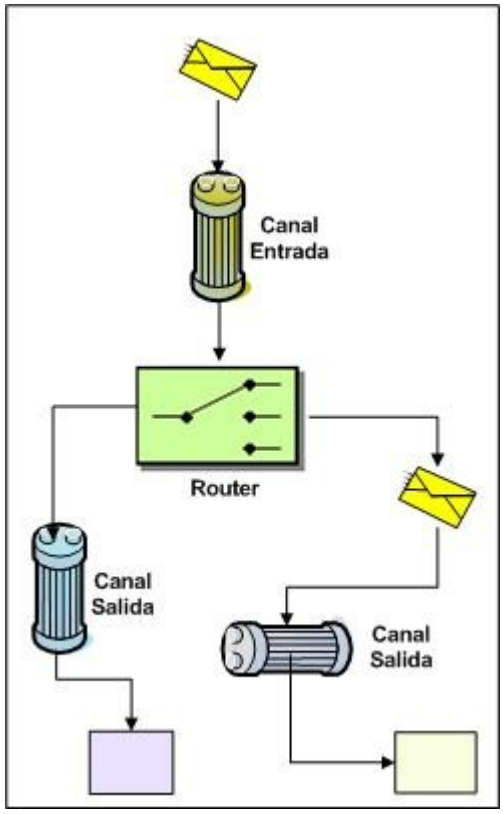


FIGURA 2.15 – Router de mensajes

El router de mensajes difiere de la noción básica de *pipes* y *filters* en que se conecta a múltiples canales de mensajes de salida.

La principal ventaja de este componente es que la decisión sobre el criterio de enrutamiento hacia el destino final se encuentra en un solo lugar.

D1. Variantes de routers

Los routers de mensajes pueden utilizar diferentes criterios para determinar el canal de salida para un mensaje entrante.

El caso más simple es un router con un único canal de entrada y un único canal de salida. Este sistema es utilizado cuando se requiere integrar 2 soluciones de integración. El router sería el nexo entre ambas soluciones.

Otros tipos de routers basan la elección del canal de salida en base a propiedades del mensaje, como pueden ser el tipo de mensaje o el valor de determinadas propiedades. Estos routers son conocidos como **routers basados en contenido**.

Existe otra variante en donde la decisión de enrutamiento se basa en condiciones del ambiente. Estos routers se denominan **routers basados en contexto**. Se utilizan para situaciones en donde se requiera, por ejemplo, balanceo de carga o recuperación ante fallas.

E. Transformación de mensajes

Los formatos de datos de las aplicaciones que participan en una solución de integración, generalmente no concuerdan.

Cambiar los formatos de datos de las aplicaciones para adherir a un formato único (como puede ser una base de datos compartida) es riesgoso, y quizás no sea posible.

Los tipos de datos y nombres de campos son parte inherente a la lógica de negocios de las aplicaciones, y cualquier cambio requiere un refactoreo que no siempre resulta económico, ni sencillo.

Sin embargo, la principal desventaja en tratar de ajustar el formato de datos de una aplicación a otra es el aumento de acoplamiento entre ambas aplicaciones, rompiendo uno de los principios claves en la integración de aplicaciones que es el débil acoplamiento.

Al depender las aplicaciones del formato de representación interno de otras, cualquier cambio a realizarse en alguna de ellas implicaría un cambio en las aplicaciones relacionadas.

Para implementar la transformación de mensajes entre aplicaciones se utiliza un filtro especial, denominado **Transformador de Mensajes**, que se ubica entre otros filtros o aplicaciones.

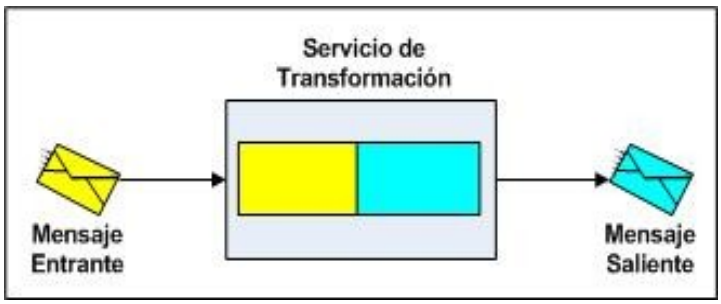


FIGURA 2.16 – Transformación de mensajes

F. *Endpoint*

Las aplicaciones y el sistema de mensajería se consideran 2 software distintos.

El sistema de mensajería se encarga de las comunicaciones entre las aplicaciones,

quienes proveen la funcionalidad requerida para algún tipo de usuario. El sistema de mensajería funciona como un servidor con capacidad para recibir pedidos de envíos de mensajes, y responder dichos pedidos.

Los clientes que utilizan los servicios de este servidor son las aplicaciones que usan la mensajería para comunicarse.

Sin embargo las aplicaciones no necesariamente conocen cómo ser clientes del sistema de mensajería.

El servidor de mensajería debe proveer de un API u otro mecanismo (específico de cada sistema) para que el cliente pueda interactuar con el servidor.

Luego, mediante determinada codificación y/o configuración, la aplicación es capaz de conectarse como cliente al sistema de mensajería, lo cual la habilita para el envío y recepción de mensajes.

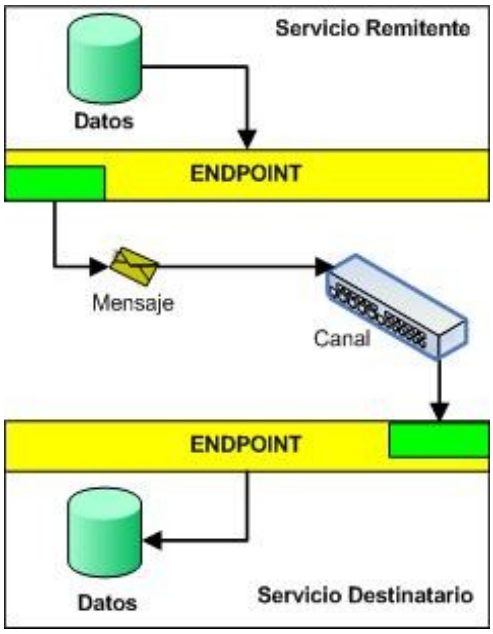


FIGURA 2.17 – *Endpoints* que conectan aplicaciones al sistema de mensajería

La aplicación no tiene conocimiento de las particularidades del sistema de mensajería.

Los datos que necesita transmitir son enviados al e*ndpoint* (FIGURA 2.17) quien se encarga de formatearlos en mensajes para ser enviados a través de canales de mensajes particulares.

Se encarga, también, de recibir los datos del canal de mensajes, extraer el contenido y prepararlos para ser entregados a la aplicación.

El e*ndpoint* abstrae a la aplicación del sistema de mensajería utilizado.

En caso de modificar el API de mensajería, o cambiar en su totalidad el sistema de mensajería, el código del e*ndpoint* debe ser modificado o reemplazado por el adecuado.

2.7. Tecnologías de Integración

2.7.1. Introducción

Tener una visión general sobre los conceptos relacionados a sistemas distribuidos nos permitirá introducir el concepto de **middleware**, importante en el desarrollo de este trabajo, y ayudará al lector a incorporar conocimientos básicos que permitirán una mejor comprensión de la problemática y posterior implementación del caso de estudio, presentado en la parte 2 de este informe.

Según George Coulouris[10]

> *"Un **sistema distribuido** es áquel en el que los componentes de hardware o software, localizados en computadoras unidas mediante una red, **comunican** y **coordinan** sus acciones mediante el paso de **mensajes**"*

Las características deseables de los sistemas distribuidos son:

- Compartir recursos
- Portabilidad
- Adaptabilidad
- Concurrencia y transaccionalidad
- Escalabilidad
- Transparencia
- Heterogeneidad

A. Compartir recursos

Uno de los principales objetivos de los sistemas distribuidos es permitir que los recursos de hardware y software sean compartidos entre los equipos que forman parte de la red.

B. Portabilidad

Esta característica implica que las aplicaciones implementadas para una solución distribuida en particular puedan ser utilizadas en otro ámbito distribuido, sin mayores inconvenientes.

C. Adaptabilidad

Los sistemas distribuidos deben ser fácilmente adaptables a modificaciones o ampliaciones de las aplicaciones intervinientes.

D. Concurrencia y Transaccionalidad

Los sistemas distribuidos deben permitir el acceso/modificación concurrente a los diferentes recursos, y manejar cuidadosamente la transaccionalidad en el acceso/modificación, para evitar errores o ambigüedad en los datos.

E. Escalabilidad

Los sistemas distribuidos deben mantener sus prestaciones ante el aumento de los recursos, aplicaciones o accesos involucrados.

F. Tolerancia a fallos

En los sistemas distribuidos los fallos son parciales, ya que aunque fallen algunos componentes, otros pueden seguir funcionando.

G. Transparencia

Esta propiedad implica el ocultamiento de detalles internos de los sistemas distribuidos, para que el sistema se perciba como una unidad, y no como un conjunto de recursos dispersos.

Hace referencia a:

- Transparencia de **acceso**

  Permite el acceso a los recursos locales y remotos de igual manera.

- Transparencia de **ubicación**

  Permite el acceso a los recursos sin necesidad de conocer su ubicación física

- Transparencia **frente a fallos**

  Esta propiedad pretende ocultar y, si es posible recuperarse, ante fallos ocurridos en el sistema distribuido.

- Transparencia de **persistencia**

  Necesario para ocultar los detalles de almacenamiento de los recursos. Si están en memoria volátil o no volátil.

- Transparencia al **escalado**

  Permite que el sistema se expanda en cuanto a sus capacidades de manera transparente a los usuarios.

H. Heterogeneidad

La variedad y diversidad en un sistema distribuido se aplica a los recursos de red, hardware, lenguajes de programación, sistemas operativos y aplicaciones heredadas.

Para que esta amplia variedad de recursos puedan funcionar de manera coordinada son necesarias ciertas capacidades: transformación de formato de datos, de protocolos, o adecuarse a protocolos estándares.

Internet, debe su amplia aceptación y uso por estar basado en protocolos estándares como TCP/IP. Sobre este protocolo hay un conjunto de servicios de más alto nivel basados en otros protocolos, HTTP, FTP, Telnet, POP3, SMTP, por mencionar algunos, que permiten realizar un conjunto de diferentes tareas.

Estos protocolos fueron ideados para una comunicación humano-computador o computador-humano. Para comunicaciones que no requieran la intervención humana, es necesaria una capa adicional e intermedia denominada **middleware**.

Básicamente, el middleware es una abstracción de la comunicación a nivel aplicación, de manera que puedan comunicarse las aplicaciones entre sí, independientemente del lenguaje de programación, sistema operativo, hardware y protocolos de red subyacentes.

La interoperabilidad entre estos middleware, y por consiguiente entre las aplicaciones que pretenden comunicar, aún no está estandarizado y no existe consenso en cuanto a las tecnologías que mejor cumplen la función de intermediarios entre aplicaciones heterogéneas.

2.7.2. Middleware

Como se ha mencionado anteriormente, en ambientes de integración generalmente se requiere el uso de múltiples tecnologías y estándares. [29]

En estos casos la interoperabilidad entre las tecnologías es un punto importante ya que éstas serán utilizadas para la implementación de la infraestructura de integración.

Lograr que las diferentes tecnologías puedan interoperar entre sí no es una tarea sencilla, incluso para tecnologías basadas en estándares abiertos.

El **middleware** es un software que ejecuta entre la capa del sistema operativo y la capa de aplicación. Conecta 2 o más aplicaciones proveyendo servicios de comunicación e interoperabilidad a las aplicaciones participantes. Permite que aplicaciones puedan interactuar entre sí, sin necesidad de tener que implementar cada una la interface adecuada para la comunicación con las demás, al adaptar y transformar la información que circula.

Este software ha obtenido cierta popularidad en los últimos tiempos, y en la actualidad, cualquier proyecto de integración debe considerar la inclusión de una o más soluciones de middleware.

El software de middleware introduce una capa de abstracción en la arquitectura de integración, pero permite reducir considerablemente la complejidad en cuanto a comunicación e interoperabilidad. Sin embargo, en determinados escenarios puede introducir cierta sobrecarga en el sistema que puede influenciar las medidas de performance, escalabilidad y otros factores de eficiencia preestablecidos.

Entre las tecnologías utilizadas por las soluciones de middleware, nombraremos las principales, y se hará un análisis más profundo de aquellas más importantes a los objetivos de este trabajo

- Tecnologías de acceso a base de datos
- Servidores de aplicaciones
- MOM (*Message Oriented Middleware*)
- RPC (*Remote Procedure Call*)
- Monitores de Transacción (*TP Monitors*)
- ORB (*Object Request Broker*)
- Servicios Web
- ESB (*Enterprise Service Bus*)

A. Tecnologías de acceso a base de datos

Las tecnologías de acceso a base de datos proveen acceso a las base de datos a través de una capa de abstracción que permite el cambio del DBMS (*Database Management System*) sin necesidad de modificaciones en el código fuente de las aplicaciones. Las tecnologías más representativas son JDBC, JDO, ODBC, ADO.NET, entre otras.

B. Servidores de aplicaciones

Los servidores de aplicaciones proveen un conjunto de servicios middleware, que juntos a herramientas de administración, permiten desplegar componentes de negocios en el contenedor.

Algunos servidores de aplicaciones tienen soporte para servicios web, MOM, ORBs, transacciones, seguridad, balanceo de carga y administración de recursos.

Los servidores de aplicaciones son una plataforma de software, es decir, una combinación de tecnologías de software necesarias para la ejecución de aplicaciones. De esta manera, los servidores de aplicaciones definen la infraestructura de las aplicaciones desarrolladas y ejecutadas sobre él.

Aunque algunos servidores de aplicaciones implementan plataformas privativas, existe una gran mayoría de otros servidores de aplicaciones que brindan soporte a plataformas abiertas, estándares y generalmente más aceptadas en la industria, como son algunas de las implementaciones de Java Enterprise Edition (JEE).

C. Message-oriented Middleware (MOM)

El MOM es una infraestructura cliente/servidor que permite la interoperabilidad, flexibilidad y portabilidad de las aplicaciones. Facilita la comunicación entre las aplicaciones sobre plataformas distribuidas y heterogéneas. Reduce la complejidad al ocultar detalles de comunicación, de las plataformas y protocolos involucrados.

La funcionalidad del MOM es accesible vía APIs que residen en ambos lados: cliente y servidor. Provee comunicación asincrónica y utiliza colas de mensajes para el almacenamiento temporario de los mensajes intercambiados.

No todos los productos MOM soportan las distintas plataformas, sistemas operativos y protocolos existentes. La plataforma Java permite lograr independencia del proveedor a través de una interface común utilizada para el acceso de productos middleware: el servicio de mensajería Java (JMS) [44]

D. Remote Procedure Call (RPC)

RPC es una infraestructura cliente/servidor que permite la interoperabilidad de aplicaciones en ambientes heterogéneos. Al igual que MOM permite la comunicación entre software sobre distintas plataformas y oculta casi todo los detalles de comunicación.

RPC está basado en conceptos procedurales y permite reducir la complejidad asociada al desarrollo de aplicaciones implementadas en lenguajes distintos, que ejecutan sobre plataformas heterogéneas y se comunican a través de diferentes protocolos de red, a partir del ocultamiento de los detalles de transporte y transformación.

RPC no puede ser considerado estrictamente un software de middleware ya que está embebido dentro de las aplicaciones cliente y servidor. Al compilarse el

cliente y servidor se crean clases *proxy*, que son utilizadas al momento de invocar un método remoto.

La principal diferencia con MOM radica en la manera de comunicarse. Mientras que MOM soporta comunicación asincrónica, RPC promueve la comunicación sincrónica, estilo *request/reply*, que bloquea el cliente hasta tanto el servidor no finalice el procesamiento.

RPC usualmente está relacionado con DCE (*Distributed Computing Environment*), desarrollado por la OSF[18] (*Open Systems Foundation*). DCE es un conjunto de servicios de integración que expanden la funcionalidad de RPC. Además de RPC, DCE provee servicios de directorio, seguridad, *threading*, entre otros.

E. Monitores de Procesamiento de Transacciones (*TP Monitors)*

Los TP son considerados la primera generación de servidores de aplicaciones. Es una tecnología middleware importante en aplicaciones de misión crítica. Están basados en el concepto de transacciones. Se encargan del monitoreo y coordinación de transacciones entre diferentes recursos, además de ofrecer servicios de seguridad y administración de la performance.

La administración de performance es lograda a partir de *pools* de recursos y balanceo de carga, que facilitan el acceso concurrente de clientes, y permiten un uso más eficiente de los recursos.

Esta tecnología era usada en escenarios donde un ambiente de soporte online debía operar concurrentemente con procesos batch.

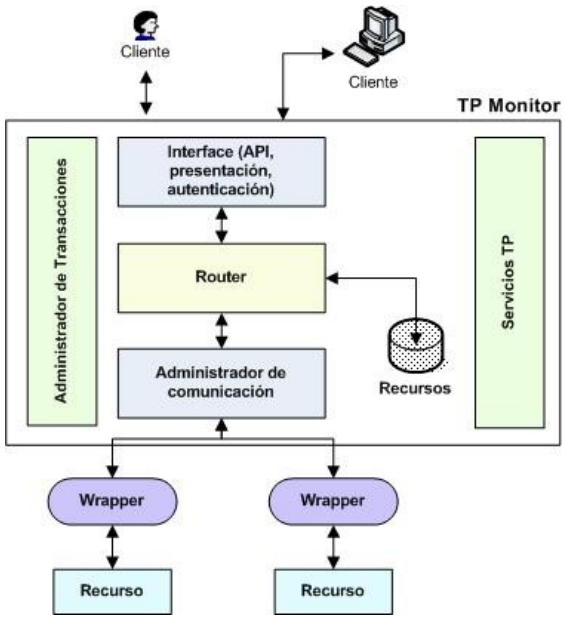


FIGURA 2.18 – Monitor de Procesamiento de Transacciones

18 **OSF:** http://es.wikipedia.org/wiki/Open_Software_Foundation

Los monitores TP tradicionalmente fueron utilizados en sistemas de información heredados. Están basados en un modelo procedural, utilizando RPC para comunicación entre las aplicaciones. Son difíciles de programar debido a la complejidad de los APIs a través de los cuales exponen su funcionalidad.

Los monitores TP son privativos, lo cual dificulta su migración a otras tecnologías middleware.

F. ORB (*Object Request Broker*)

ORB es una tecnología middleware que administra la comunicación entre objetos distribuidos o componentes. Los detalles de implementación de los ORBs no son visibles lo cual permite brindar transparencia de ubicación, de lenguaje de programación, protocolo y sistema operativo.

La comunicación entre los objetos distribuidos es usualmente sincrónica, y está basada en interfaces. Esta última característica es importante ya que facilita el mantenimiento de los componentes al estar ocultos los detalles de implementación.

Los tres estándares ORB más importantes son:

- OMG CORBA ORB
- Java RMI y RMI/IIOP
- Microsoft COM/DCOM/COM+ / .NET Remoting/WCF

Los productos compatibles con RMI/IIOP y CORBA ORB son interoperables al utilizar el mismo protocolo (IIOP) para comunicar los componentes entre sí.

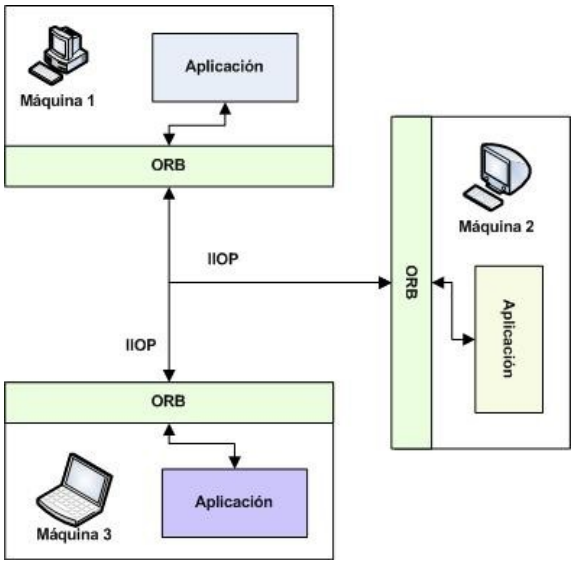


FIGURA 2.19 – Interacción entre objetos mediante ORB

Existen distintas implementaciones de ORBs que permiten compilarlo junto al cliente, ejecutarlo como un proceso separado dentro del S.O, o bien embeberlo dentro del *kernel* del S.O.

Las grandes limitaciones sobre ORB, están relacionadas con la falta de interoperabilidad con otras tecnologías y costos de adquisición de algunas implementaciones.

G. Servicios Web

Los servicios web son considerados el siguiente paso en la evolución de arquitecturas distribuidas. Ofrecen la base tecnológica para lograr la interoperabilidad entre aplicaciones utilizando diferentes plataformas de software, sistemas operativos y lenguajes de programación [21][29]

Los servicios web son la primer tecnología que ha tenido mayor aceptación entre la mayoría de los proveedores de software, permitiendo de esta manera lograr altos niveles de interoperabilidad entre aplicaciones ejecutando sobre plataformas heterogéneas.

Están basados en 3 especificaciones fundamentales: SOAP[19], WSDL[20], UDDI[21].

Los servicios web difieren de otras tecnologías distribuidas al soportar débil acoplamiento a través de operaciones que intercambian únicamente datos. Los modelos componentes y objetos distribuidos también pueden intercambiar comportamiento.

Los servicios web proveen soporte para interacciones asincrónicas y sincrónicas. No tienen estado y utilizan protocolos de internet estándares, tales como HTTP, FTP, SMTP y MIME.

Sin embargo, los servicios web básicos tienen ciertos inconvenientes, algunos relacionados con performance respecto a soluciones similares que utilizan protocolos binarios para la comunicación, y otros con características de QoS (*Quality of Service*) tales como seguridad o transaccionalidad, que otros modelos de componentes proveen.

Esta falta de características de QoS son subsanadas a partir de un conjunto de especificaciones adicionales -las especificaciones WS-*, tales como: WS-Security, WS-Coordination, WS-Addressing, WS-Policy, entre otras.

H. Enterprise Service Bus (ESB)

El ESB es un software de infraestructura que actúa como una capa intermedia

19 **SOAP**: http://es.wikipedia.org/wiki/SOAP

20 **WSDL:** http://es.wikipedia.org/wiki/WSDL

21 **UDDI:** http://es.wikipedia.org/wiki/UDDI

que brinda servicios de integración entre servicios web y otras tecnologías de middleware, seguridad, administración, mientras controla las comunicaciones entre los distintos componentes.

El ESB funciona como un mediador entre diferentes, y generalmente incompatibles, protocolos y productos middleware.

Provee una infraestructura de comunicación escalable, segura y robusta para que los servicios puedan comunicarse entre sí, además de ofrecer características avanzadas en el control de servicios, y capacidad para interceptar y procesar los mensajes que circulan por el bus.

También ofrecen servicios de infraestructura para el enrutamiento y transformación de mensajes antes de ser entregados al servicio destino.

2.8. Patrones de diseño en integración de aplicaciones

2.8.1. Introducción

Los patrones de diseño en ingeniería de software describen métodos conocidos de resolución de problemas recurrentes. No deben ser considerados como una solución "*lista para usarse*", sino como una plantilla que puede ser reutilizada en diferentes situaciones.

Usados adecuadamente los patrones de integración pueden ayudar a disminuir la brecha existente entre la visión global y abstracta de la integración y la implementación real del sistema.

El diseño e implementación de aplicaciones puede resultar en una tarea compleja y dificultosa. Existen numerosas tecnologías, ambientes de ejecución, plataformas de software y hardware, pero usualmente los problemas a resolver son similares.

Los patrones de diseño ofrecen maneras probadas de resolver tales problemas.

Existen patrones de diseño específicos para los sistemas de mensajería e integración de aplicaciones. En esta parte se analizarán los principales patrones conocidos, algunos de los cuales serán utilizados posteriormente como guía en la implementación del caso de estudio [8][14][52]

2.8.2. Patrones

A. Patrón: Canales de Mensajería

Las variantes más conocidas de este patrón son:

- Datatype channel
- Point-to-Point channel
- Publish-Subscribe Channel

Estas variantes realizan diferentes tareas, pero todas están basadas en el mismo patrón

A.1. Datatype Channel

Notación

(fuente: Enterprise Integration Patterns[87])

Descripción

Utilizado por las aplicaciones que necesitan transmitir a través de un sistema de mensajería diferentes tipos de datos

A.2. Point-to-Point Channel

Notación

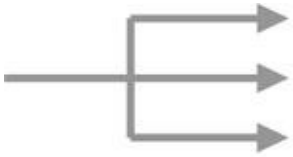
(fuente: Enterprise Integration Patterns[87])

Descripción

Utilizado por las aplicaciones que realizan invocaciones a procedimientos remotos, o transfieren documentos a través de un sistema de mensajería.

A.3. Publish-Subscribe Channel

Notación



(fuente: Enterprise Integration Patterns[87])

Descripción

Utilizado por las aplicaciones que necesitan anunciar eventos por el sistema de mensajería

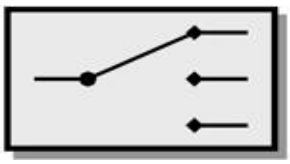
B. Patrón: Ruteo de Mensajes

Las variantes más conocidas de este patrón son:

- Content-based Router
- Message Filter
- Dynamic Router
- Recipients List
- Splitter
- Aggregator
- Routing Slip
- Process Manager
- Message Broker

Content-based Router

Notación

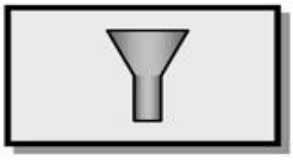
 (fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón es utilizado por sistemas que requieren leer el contenido de los mensajes para, junto a ciertas reglas de ruteo, dirigir los mensajes a los destinatarios adecuados.

Message Filter

Notación

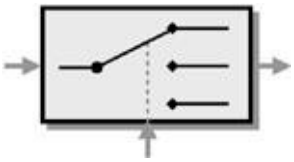
 (fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón tiene un funcionamiento similar al patrón anterior. Lee el contenido del mensaje, y si concuerda con cierto criterio deja que el mensaje continúe su trayecto. Caso contrario lo descarta

Dynamic Router

Notación



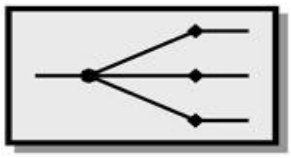
(fuente: Enterprise Integration Patterns[87])

Descripción

Es una variante más flexible que el router basado en contenido, al permitir que las reglas de ruteo sean modificadas dinámicamente a través de mensajes de control.

Recipients List

Notación



(fuente: Enterprise Integration Patterns[87])

Descripción

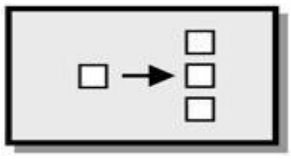
Este patrón extiende la funcionalidad provista por el router basado en contenido.

Funciona de manera similar al patrón *Publish-Subscribe channel*, al inspeccionar el mensaje entrante y en base a su contenido determina la lista de destinatarios.

Luego reenvía el mensaje a esos destinatarios.

Splitter

Notación



(fuente: Enterprise Integration Patterns[87])

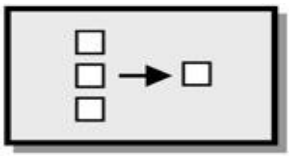
Descripción

Este patrón es usado en situaciones en que el mensaje entrante contiene múltiples elementos que no pueden ser procesados todos de la misma manera.

En este caso el mensaje es dividido en elementos separados y cada uno de ellos es enviado independientemente a los subsistemas apropiados para su procesamiento.

Aggregator

Notación

 (fuente: Enterprise Integration Patterns[87])

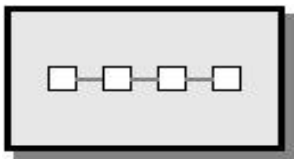
Descripción

Este patrón es usado en situaciones opuestas a la descripta en el patrón anterior. Se reciben mensajes entrantes, y se identifican aquellos que están correlacionados.

Cuando ya se recibieron todos los mensajes correlacionados se colecta el contenido de cada uno de estos mensajes, se los agrega y se publica un nuevo y único mensaje.

Routing Slip

Notación

 (fuente: Enterprise Integration Patterns[87])

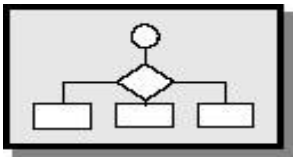
Descripción

Este patrón es usado cuando se requiere conocer el camino completo que recorrerá un mensaje.

Cada mensaje entrante tendrá asociado la secuencia de pasos de procesamiento que necesitará atravesar.

Process Manager

Notación

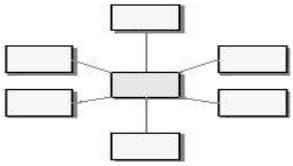
 (fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón se utiliza en situaciones similares a la descripta en el patrón anterior. Sin embargo funciona de una manera más dinámica. Reenvía el mensaje a la primer unidad de procesamiento, y en base al resultado generado e información sobre el procesamiento realizado, determina el próximo paso de procesamiento.

Message Broker

Notación



(fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón es esencial en muchas estrategias de integración. Permite conectar todos los sistemas involucrados en la estrategia de integración. Internamente utiliza los patrones antes mencionados, para realizar ruteos de mensajes entre los sistemas conectados. Además reduce el número de canales de mensajes necesarios para comunicar los sistemas entre sí.

C. Patrón: Transformación de Mensajes

Las variantes más conocidas de este patrón son:

- Envelope Wrapper
- Content Enricher
- Content Filter
- Normalizer
- Message Translator

Envelope Wrapper

Notación

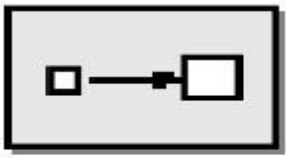
(fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón se utiliza en ocasiones donde el sistema de mensajería requiere que los mensajes tengan un formato predeterminado. El destinatario del mensaje descarta el "envoltorio" y recupera el mensaje tal cual fue enviado por el remitente.

Content Enricher

Notación



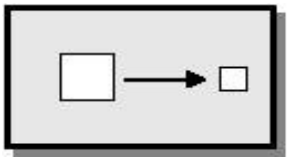
(fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón es usado en situaciones que el destinatario del mensaje requiere más información que la que el remitente puede proveer. En estos casos es necesario "enriquecer" el mensaje original con información adicional recuperada de fuentes de información externas.

Content Filter

Notación



(fuente: Enterprise Integration Patterns[87])

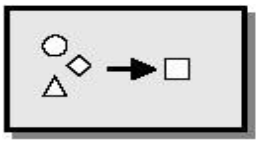
Descripción

Este patrón es usado en situaciones opuestas al patrón descripto anteriormente.

Cuando un mensaje entrante contiene información compleja y solamente una pequeña parte de esa información es requerida por el destinatario, este patrón remueve del mensaje lo que no es necesario.

Normalizer

Notación



(fuente: Enterprise Integration Patterns[87])

Descripción

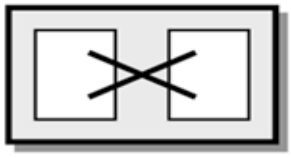
Este patrón es una combinación de un router de mensajes y múltiples transformadores de mensajes. Es útil cuando los sistemas integrados utilizan diferentes formatos de mensajes y cuando cada uno de esos formatos requiere un tipo de traducción diferente de manera de poder adecuarse al modelo usado por el sistema de mensajería.

La información puede llegar en formatos tales como XML, texto plano, planillas de cálculo, archivos separados por coma, mensajes SOAP, entre otros. Cada uno de dichos formatos requiere un procesamiento diferente de manera de transformarlo

al formato apropiado para el sistema de mensajería usado.

Message Translator

Notación

 (fuente: Enterprise Integration Patterns[87])

Descripción

Este patrón es necesario cuando los formatos de datos de las aplicaciones que necesitan integrarse no coinciden. Aunque es posible modificar el formato de alguna de las aplicaciones para concordar con las demás aplicaciones, ésto generalmente es dificultoso y poco viable en determinados escenarios.

Este patrón se utiliza entonces en los escenarios de integración en los cuales es necesario transformar los formatos de los datos de los sistemas que están interactuando

Capítulo 3

SOA (Service Oriented Architecture)

3.1. Desafío en la construcción de aplicaciones en el nuevo “Mundo Conectado”

En los últimos años el área del software ha avanzado y evolucionado en conceptos y tecnologías usadas para construir sistemas de información que se adapten al nuevo **"mundo conectado".**

Las bases de esta evolución fueron establecidas por SOA, y algunos conceptos relacionados: tecnología de infraestructura de comunicación, composición de servicios, reusabilidad, integración. Es decir, conceptos y herramientas que ofrecen la capacidad de construir e integrar software heterogéneo a partir de componentes reusables, autónomos, adaptables y escalables, basados en tecnologías estándares.

En este nuevo “mundo conectado” una de las claves es la **flexibilidad**. Aquellas organizaciones con mayor capacidad de cambio ante factores externos, serán las más aptas para la supervivencia.

Los procesos y sistemas se van complejizando, siendo desplazada la etapa donde la automatización de tareas era responsabilidad de sistemas individuales, por otra donde todos los sistemas forman parte de un gran sistema distribuido.

En este contexto, la **mantenibilidad** de tales sistemas distribuidos es otro factor importante a tener en cuenta.

Otras prácticas que no funcionan más en este nuevo mundo, son aquellas que enfrentaban los problemas de escalabilidad y distribución. Ya no funciona el sistema de control centralizado. En los ambientes IT actuales las aplicaciones que ejecutan sobre plataformas heterogéneas pueden necesitar comunicarse e

integrar funcionalidad entre sí, de manera no intrusiva. Son necesarias prácticas que promuevan la **descentralización** y **heterogeneidad**.

SOA se erige como la respuesta a tales cuestiones. Garantiza la escalabilidad y flexibilidad de los sistemas mientras éstos siguen creciendo. [13][15][30]

Básicamente, SOA se apoya en tres elementos:

- **Servicios** que representan las unidades autocontenidas que encapsulan las funcionalidades de negocios requeridas, y que pueden ser implementadas por distintas tecnologías, para diversas plataformas.
- Una **infraestructura** específica, de la cual el ESB forma parte, y permite combinar los nombrados servicios de manera fácil y estándar.
- **Políticas y procesos** que permitan hacer frente a la heterogeneidad de los sistemas distribuídos, a la descentralización y tolerancia a fallos.

Sin embargo SOA no es un producto *por sí mismo*, sino un paradigma, una forma de pensar, que debe ser construida, diseñada cuidadosamente para cumplimentar sus objetivos.

La siguiente frase quizás refleje la idea u objetivo primordial de SOA, traspolando su mensaje al ámbito tecnológico.

> *"No es la especie más fuerte que sobrevive, ni siquiera la más inteligente, sino aquellas con mejor capacidad de adaptación al cambio"*[22]

3.1.1. "Orientación a servicios"

Debido al tiempo de existencia del término "***orientado a servicios***", éste fue usado en distintos contextos y para diferentes propósitos, principalmente para referirse a una manera diferente de concretar la separación de aspectos. Lo cual significa que la lógica requerida para resolver un problema puede ser construída, diseñada, codificada y administrada si se descompone en un conjunto de pequeñas piezas relacionadas entre sí. Cada una de estas piezas se encarga de un aspecto específico del problema.

Cuando se agrega "*arquitectura*", la "*orientación a servicios*" adquiere una connotación técnica. Arquitectura orientada a servicios (SOA) es un término que representa un modelo en el cual la lógica se descompone en diferentes y pequeñas unidades, que colectivamente comprenden una pieza mayor de lógica de negocios. Individualmente estas piezas pueden ser distribuidas.

SOA, de esta manera, promueve que estas unidades individuales de lógica se adecuen a principios generales de la arquitectura, pero manteniendo su

22 Frase correspondiente a Charles Darwin

autonomía, de manera que les permita evolucionar independientemente unas de otras. En el contexto de SOA estas unidades de lógica de negocios son conocidas como **servicios**.

3.1.2. Principios de la orientación a servicios

No existe aún consenso en cuanto a cuales son los principios de la orientación a servicios, por lo tanto, lo único que se puede enumerar es un conjunto de principios que están muy asociados con la orientación a servicios.

Estos principios según Thomas Erl [13] son:

A. Los servicios deben ser **reusables**

B. Los servicios deben proveer de un **contrato formal**

C. Los servicios deben ser **débilmente acoplados**

D. Los servicios deben permitir la **composición**

E. Los servicios deben ser **autónomos**

F. Los servicios **no** deben tener **estado**

G. Los servicios deben poder ser **descubiertos**

A. Reusabilidad de servicios

La reusabilidad es un concepto simple de comprender, pero generalmente no resulta fácil su implementación.

- Programas de único propósito

  Construir un programa para un único propósito permite enfocarse solamente en un conjunto específico de requerimientos. El programa puede ser optimizado y adaptado para cumplimentar sus objetivos en los escenarios conocidos donde ejecutará.

  Su acotado alcance afecta sólo las partes del programa que forman parte del ciclo de desarrollo. Su diseño, implementación y *testing* son más sencillos ya que el alcance de su uso es limitado y predecible. Su despliegue y administración no son mucho más complejo nuevamente, ya que solamente se necesita asegurar que cumpla con su único propósito.

- Programas multi-propósito

  La construcción de programas útiles para más de un propósito requiere de ciertas consideraciones, no contempladas en el caso anterior.

  Es necesario determinar cómo serán utilizados estos programas en múltiples y diferentes escenarios de uso; ésto conlleva a la necesidad

de ofrecer funcionalidad más genérica, con un rango amplio de funciones ofrecidas, que implica diseños complejos, con esfuerzos de desarrollo mayores.

Una vez que el programa ha sido implementado y se encuentra en uso, ya sea por humanos u otras computadoras, se pierde la libertad de realizar cambios arbitrarios. Sus clientes se han convertido en dependientes de él, especialmente si su uso es a través de un API.

Los anteriores son algunos de los factores que tradicionalmente limitaron la posibilidad de tener éxito en lograr la reusabilidad de aplicaciones.

Las consideraciones mencionadas en el caso de aplicaciones multi-propósito son tan antiguas como la construcción de software para la venta comercial.

Aquel software construido para ser vendido públicamente ha sido diseñado desde un principio con la idea de reuso en mente.

El reuso implica agregar complejidad, incrementar costos, esfuerzos y tiempos en la construcción del software. Por lo tanto, la reusabilidad no es una práctica de diseño habitual en las organizaciones para el desarrollo de soluciones internas. Ésto condujo a la noción de aplicaciones *“como silos”*

La llegada de la orientación a objetos impulsó la idea de los potenciales beneficios que pueden obtenerse al construir aplicaciones distribuídas a partir de componentes (objetos) capaces de ser utilizados para más de un propósito.

El reuso a través del diseño orientado a objetos obtuvo cierta popularidad, con diferentes niveles de éxito, aunque surgieron determinados inconvenientes.

- El reuso de componentes estaba limitado a ambientes de ejecución y/o programas clientes propietarios
- Los componentes tenían dependencias fuertes con otros componentes
- Los componentes reusables tenían demasiada funcionalidad, usualmente no requerida.

A pesar de los inconvenientes anteriores, y experiencias no muy satisfactorias, los proveedores y organizaciones de estándares continuaron trabajando en la creación de recursos empresariales compartibles.

El advenimiento de la plataforma tecnológica de servicios web fue recibido como un paso fundamental en la búsqueda de un framework de comunicación neutral en cuanto al proveedor, que ayude a incrementar el potencial de reuso de lógica de aplicación.

La lógica de aplicación publicada como servicio web puede ser accesible a cualquier parte de la organización que tenga soporte para la tecnología de servicios web. De esta manera, el no tener limitaciones tecnológicas impuestas por el framework de servicios web facilita el incremento de

potenciales programas consumidores.

Sin embargo la sóla innovación tecnológica es insuficiente para dar solución a los problemas que surgen en la búsqueda de implementar el principio de reusabilidad.

La orientación a servicios ayuda a enfrentar tales inconvenientes proveyendo de principios que preparan los servicios para ser reusables desde su concepción.

B. Los servicios deben proporcionar un contrato formal

Un contrato, en contextos SOA, tiene un significado más amplio que simplemente una interface técnica. Establece los términos de compromiso, provee limitantes técnicas, requerimientos, y cualquier otra información semántica que el proveedor del servicio desee hacer pública.

Un contrato de servicio puede consistir de un grupo de documentos descriptores de servicio, cada uno de los cuales describe una parte del servicio.

Un contrato de un servicio web puede estar comprendido de los siguientes documentos descriptores:

- ✔ WSDL
- ✔ XSD
- ✔ WS-Policy

En el pasado, los contratos eran comúnmente representados por una forma de interface técnica conocida como API (*Application Programming Interface*). Una API podía ser accedida por una aplicación cliente instalada en la misma máquina que la librería, o bien remotamente. La última variante es más común en arquitecturas distribuídas, donde los componentes requieren de representaciones locales de los contratos (*proxies*) para interactuar con componentes ubicados en diferentes servidores.

IDL (*Interface Definition Language*) y ASN.1 (*Abstract Syntax Notation 1*) fueron usadas con frecuencia para expresar contratos técnicos para frameworks de invocación remota, tales como aquellos basados en RPC (*Remote Procedure Call*).

Los servicios web establecieron un framework de comunicaciones distribuidas no propietario que introdujo WSDL (*Web Services Definition Language*) como parte esencial de un contrato técnico.

Íntimamente relacionado con WSDL se encuentra XSD que facilita la definición del modelo de datos para el intercambio de mensajes a través del uso de servicios web, y el lenguaje WS-Policy a partir del cual es posible la definición de políticas que pueden adosarse a varias partes del documento WSDL.

C. Acoplamiento de los servicios

El término “acoplamiento”, según el diccionario español[23], refiere a la

> *“unión de dos piezas o cuerpos que se ajustan perfectamente”*

En ambientes de computación cualquier parte “separable” tiene el potencial para acoplarse a otra, a fin de agregar valor a sus objetivos.

Una de las maneras más comunes de explicar acoplamiento es compararlo con **dependencia**. El acoplamiento existente entre dos “partes” es equivalente al nivel de dependencia existente entre ellas.

Comúnmente, en arquitecturas anteriores, el acoplamiento entre las aplicaciones o componentes era alto.

En las tradicionales arquitecturas cliente/servidor de 2 capas, los clientes eran diseñados para interactuar específicamente con cierta base de datos, o proceso del lado del servidor. Los comandos necesarios para las conexiones entre el cliente y servidor estaban embebidos en los clientes, por lo tanto, cualquier cambio requería la recompilación de todas las instalaciones existentes.

El acoplamiento de servicios está dado por la relación que existe entre el servicio, el ambiente subyacente y sus consumidores.

El objetivo de este principio es promover el nivel adecuado de acoplamiento de los servicios, de manera que continúen siendo un recurso útil y accesible, mientras que se los protege junto a sus consumidores, de crear una relación que inhiba o restrinja la posibilidad de evolución de cada uno independientemente del otro.

El contrato de servicio es el principal elemento sobre el cual giran la mayoría de las consideraciones respecto al acoplamiento. El principal aspecto a analizar es la relación entre el contrato de servicio y la lógica que el mismo encapsula.

La FIGURA 3.1 muestra las dos opciones de acoplamiento existentes. Es posible diseñar el contrato con dependencias en la lógica subyacente del servicio, o bien es posible diseñar la lógica del servicio con dependencias en el contrato

FIGURA 3.1 - Contrato de servicio y acoplamiento

23 **Definición Acoplamiento:** http://www.wordreference.com/definicion/acoplamiento

C.1. Acoplamiento lógica a contrato

La opción recomendada para la construcción de servicios es diseñar primero su contrato, y posteriormente la lógica subyacente.

Este proceso conocido como ***contract-first*** puede resultar en cierto acoplamiento entre el contrato de servicio y la lógica de negocios (conocido como *acoplamiento lógica a contrato*) ya que ésta última fue creada específicamente para soportar el contrato diseñado independientemente (FIGURA 3.2).

Los servicios web, por ejemplo, creados mediante este proceso, resultarán acoplados al contrato de servicio, usualmente creados mediante documentos WSDL.

Sin embargo, debido a que el contrato no está acoplado a la lógica subyacente, ésta puede ser modificada o cambiada en su totalidad sin que se vean afectados los consumidores del servicio web que han generado una dependencia únicamente del contrato de tal servicio.

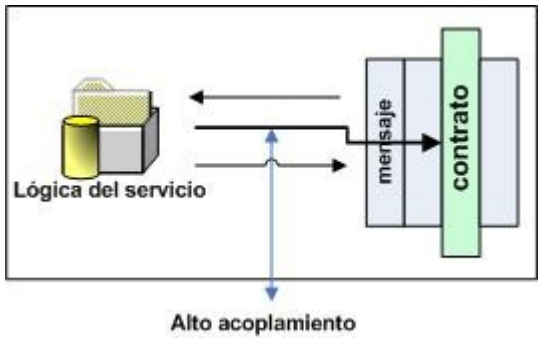


FIGURA 3.2 – Alto acoplamiento

A pesar que este tipo de **acoplamiento** es considerado **positivo** y una manera adecuada de construir servicios de acuerdo a los principios de orientación a servicios, sólo puede aplicarse en casos que los servicios sean construidos desde cero.

En ambientes donde los servicios son *wrappers* de aplicaciones heredadas, el tipo de acoplamiento es diferente.

jESBihca utiliza esta técnica para la generación automática, a partir del documento WSDL, de la lógica de negocios asociada al escenario de búsqueda basado en mensajes SOAP (FIGURA 3.3)

```
<plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>${cxf-version}</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <configuration>
                <sourceRoot>${basedir}/target/jaxws</sourceRoot>
                <wsdlOptions>
                    <wsdlOption>
                        <wsdl>${basedir}/src/main/resources/wsdl-search.wsdl</wsdl>
                        <extraargs>
                            <extraarg>-verbose</extraarg>
                        </extraargs>
                    </wsdlOption>

                </wsdlOptions>

            </configuration>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

FIGURA 3.3 – Configuración generación automática de código a partir de WSDL

C.2. Acoplamiento contrato a lógica

La sección anterior describió el tipo de acoplamiento donde el diseño del contrato de servicio se realiza antes que el desarrollo de la lógica de negocios.

De esta manera se logra mayor independencia del contrato respecto a la lógica, maximizando la posibilidad que el servicio evolucione independientemente de sus consumidores.

Sin embargo, en ocasiones es necesario derivar el contrato de la lógica de negocios preexistente.

Este caso es contrario al analizado anteriormente, siendo el contrato dependiente de la implementación de la lógica subyacente.

Esta dependencia del contrato respecto a la lógica de negocios se conoce como *“acoplamiento contrato a lógica”* (FIGURA 3.4)

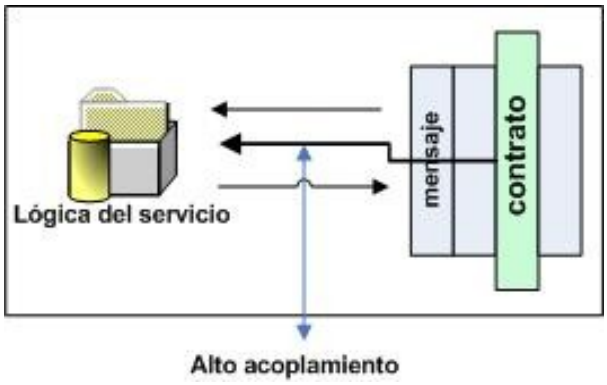


FIGURA 3.4 – El contrato de servicio está altamente acoplado a la lógica de negocios

El ejemplo más común de este escenario es la autogeneración de definiciones WSDL a partir de código fuente.

En este caso el **contrato** del servicio queda **acoplado** a las características de su ambiente de implementación.

Esta técnica es considerada un **anti-patrón** que inhibe la evolución independiente del servicio, y requiere modificaciones en los consumidores ante cambios en la lógica de negocios, y en consecuencia del contrato de servicio.

D. Los Servicios deben permitir la composición

La **reusabilidad** es un concepto esencial en la orientación a servicios; la capacidad de agregar y combinar repetidamente servicios está relacionada con el éxito de su realización.

Ensamblar funcionalidades desde diferentes fuentes forma la base de la computación distribuida.

Este principio introduce, además, nuevas consideraciones de diseño que aseguren que los servicios sean capaces de participar en múltiples composiciones que permitan la resolución de problemas mayores (FIGURA 3.5)

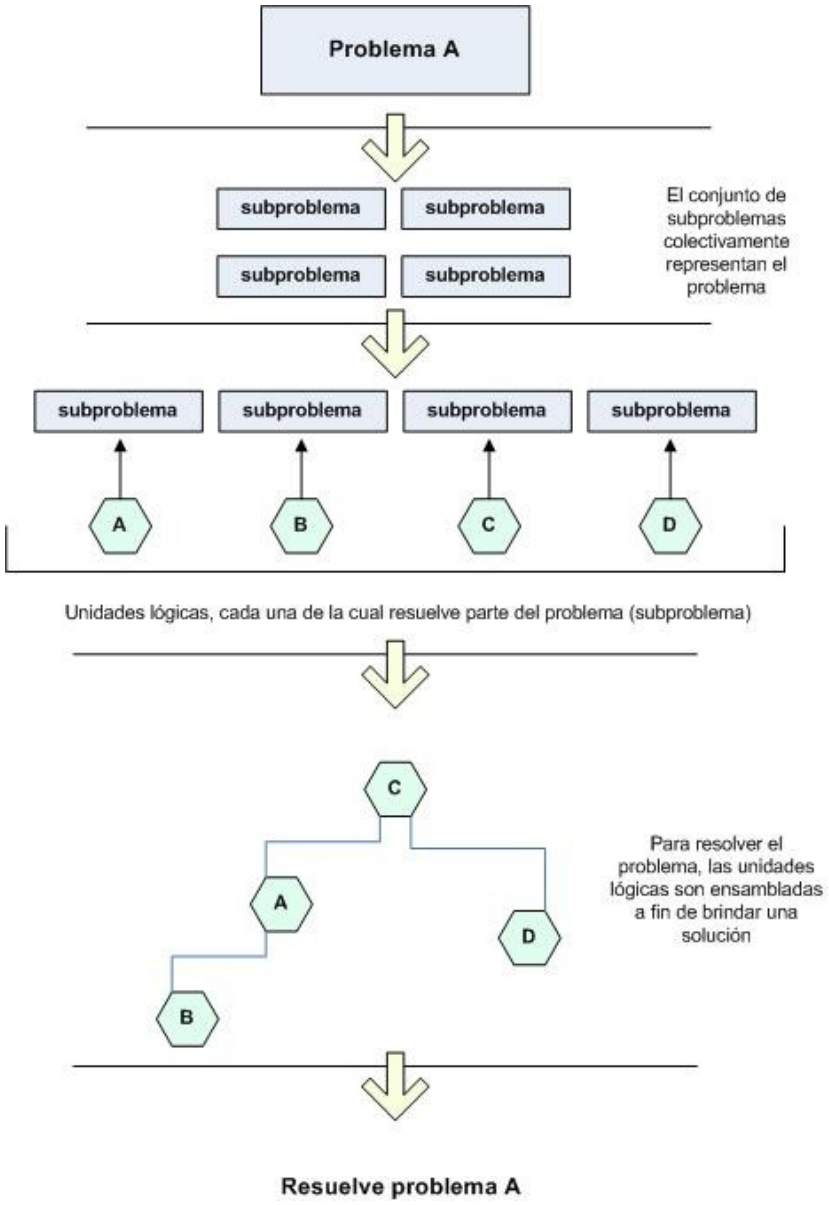


FIGURA 3.5 – Composición de funcionalidad

El diseño orientado a objetos formalizó la noción de descomponer y componer funcionalidad.

La lógica de programación podía ser separada en clases, cada una ofreciendo una funcionalidad bien específica, que luego podían ser agregadas para formar clases más complejas (FIGURA 3.6)

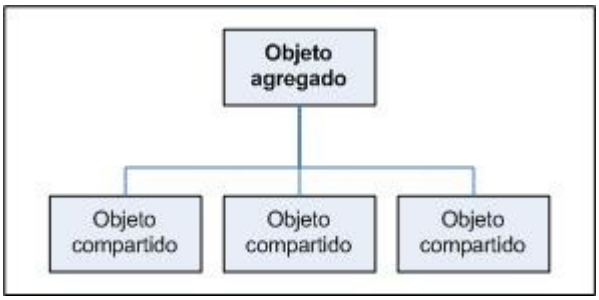


FIGURA 3.6 – Objeto agregado

Este principio está muy relacionado con otros principios del paradigma de orientación a servicios, e incluso, algunos de tales principios existen para brindar soporte a este principio. Es así que este principio comparte la mayoría de los objetivos del principio de reusabilidad. La composición de servicios es posible considerarla como una forma de reuso de servicios.

En **jESBihca** la **composición de servicios** es implementada agregando servicios individuales (*service units*, en terminología JBI), en un elemento con un formato estándar (*service assemblies*, en terminología JBI) que se empaqueta y despliega en el contenedor JBI (FIGURA 3.7)

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>esbusecase-providers-search-soap-sa</name>
      <description>ESB : PROVIDERS : SEARCH SOAP : CXF - SA</description>
    </identification>
    <service-unit>
      <identification>
        <name>esbusecase-providers-search-soap-se-su</name>
        <description>ESB :: USECASE</description>
      </identification>
      <target>
        <artifacts-zip>esbusecase-providers-search-soap-se-su-1.0.zip</artifacts-zip>
        <component-name>servicemix-cxf-se</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

FIGURA 3.7 – Configuración de un *service assembly* (SA)

E. Los Servicios deben de ser autónomos

La autonomía representa la capacidad de un servicio de autogobernarse. Significa

que tiene el control y libertad de tomar decisiones sin la influencia de elementos externos.

Luego el nivel de autonomía puede calcularse en base a la capacidad del servicio de actuar independientemente del contexto o ambiente donde ejecuta (FIGURA 3.8)

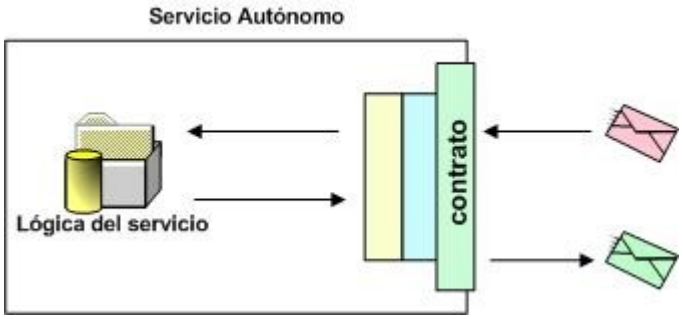


FIGURA 3.8 - Servicio autónomo

Este principio, básicamente permite incrementar la confiabilidad, predecibilidad, especialmente al ser reusados o compuestos con otros servicios, y el control que tiene el servicio sobre el ambiente donde ejecuta.

Existen algunos aspectos generales de SOA que se simplifican mediante la aplicación de este principio.

- La habilidad de escalar de un servicio en respuesta a una mayor demanda de uso
- La opción de modificar u optimizar el ambiente de ejecución de los servicios
- La libertad de aumentar, optimizar o reemplazar la tecnología de un servicio en respuesta a nuevos requerimientos

F. Los Servicios no deben tener estado

Un buen indicador que un servicio es agnóstico es el nivel de reuso y participación en composiciones que tiene. Este resultado enfatiza la necesidad de optimizar la lógica de procesamiento de los servicios a fin de soportar los requerimientos de múltiples consumidores, mientras el servicio en sí mismo consume la menor cantidad de recursos posible.

A medida que la composición se complejiza, también lo hacen los datos que necesitan ser administrados y retenidos durante la vida útil de la composición.

Los servicios requieren procesar y mantener estos datos esperando que otros servicios de la composición concluyan con su lógica de procesamiento puede afectar la infraestructura general.

Este principio promueve en el diseño de servicios la delegación en la responsabilidad de administración de estado, de manera de mantener los servicios en la condición de no tener estado cuando sea posible. (FIGURA 3.9)

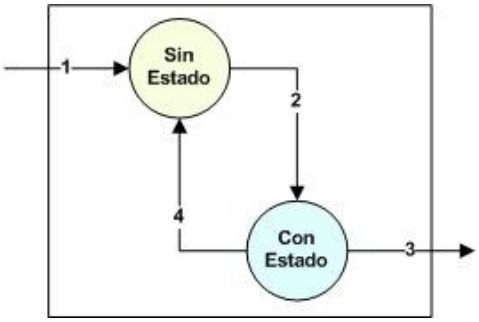


FIGURA 3.9 – Estado de los servicios

F.1. Administración de estado

Estado se refiere a una condición general de algo. Por ejemplo, una persona al caminar se encuentra en "*estado de movimiento*", mientras que al descansar se encuentra en "*estado de reposo*". Lo mismo ocurre en el ámbito IT. Los programas de computación atraviesan diferentes estados debido, usualmente, a las características del ambiente de ejecución.

Cada estado puede ser representado y descripto por datos que típicamente tienen una vida útil equivalente a la duración en la cual el programa se mantiene activo para una determinada tarea.

Por lo tanto, todas las variaciones de la información referente a estados tienden a ser temporarias por naturaleza.

La administración de estado, entonces, es considerada la administración temporal de datos específicos de una tarea determinada.

F.2. Orígenes de la administración de estado

En las antiguas soluciones 2 capas, los clientes usualmente retenían en memoria gran cantidad de datos, por largos períodos de tiempo (FIGURA 3.10).

Ésto no era considerado un problema ya que cada programa cliente era desplegado en una computadora dedicada, para ser usada por solamente un cliente.

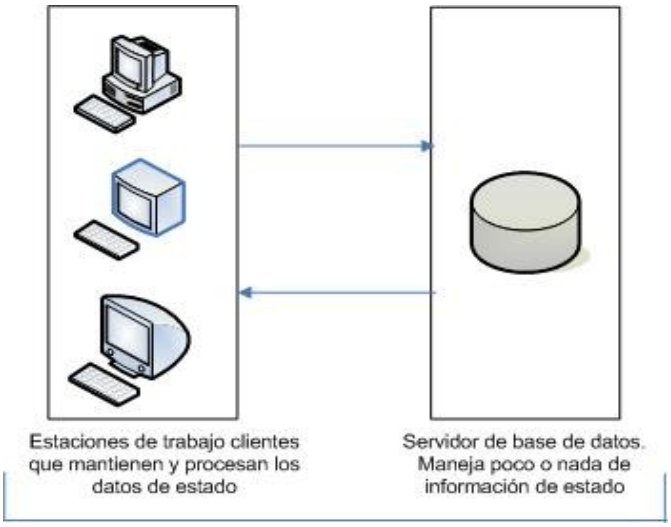


FIGURA 3.10 – Clientes que mantienen gran cantidad de datos de estado

En el modelo de computación distribuida, el procesamiento de la lógica de aplicación se movió desde la estación de trabajo cliente, a una capa intermedia.

Como resultado, la parte servidora necesitaba administrar la interacción con múltiples clientes, cada uno de los cuales tenía un requerimiento particular de procesamiento de la información de estado (FIGURA 3.11).

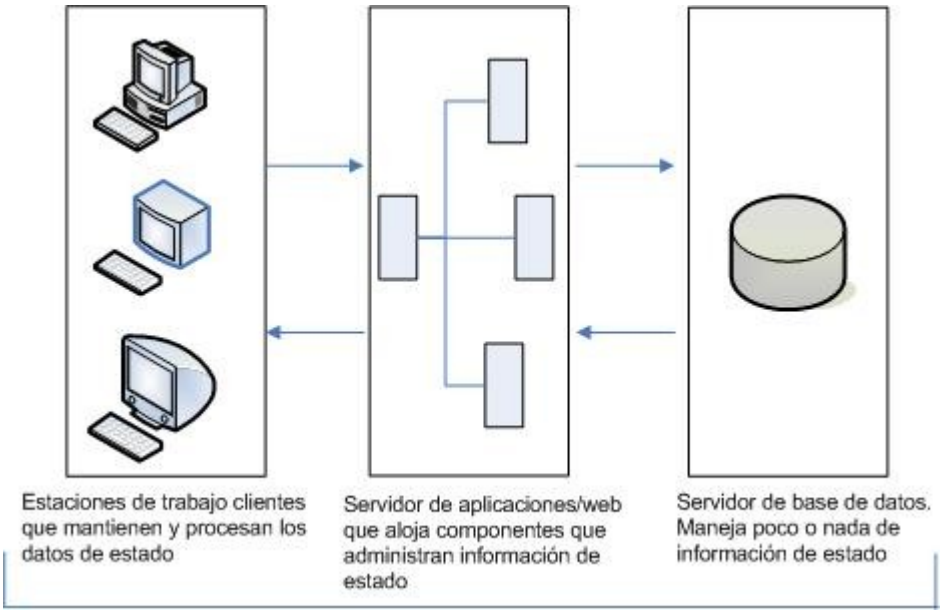


FIGURA 3.11 – Múltiples clientes y capa intermedia

Procesar y retener información de estado requiere consumo de memoria y ciclos de CPU. Un programa del lado del servidor que atienda múltiples pedidos

concurrentes puede fácilmente aumentar estas cantidades.

Para dar respuesta a este riesgo que puede incidir en cuestiones de performance, existen extensiones a las tradicionales arquitecturas distribuidas que proveen un mecanismo de delegación del procesamiento de información de estado.

Principalmente las extensiones se han centrado en opciones que ofrecen almacenamiento de datos alternativo. Una base de datos, por ejemplo, es posible que sea utilizada por los componentes para persistir la información de estado, que luego puede ser recuperada (FIGURA 3.12)

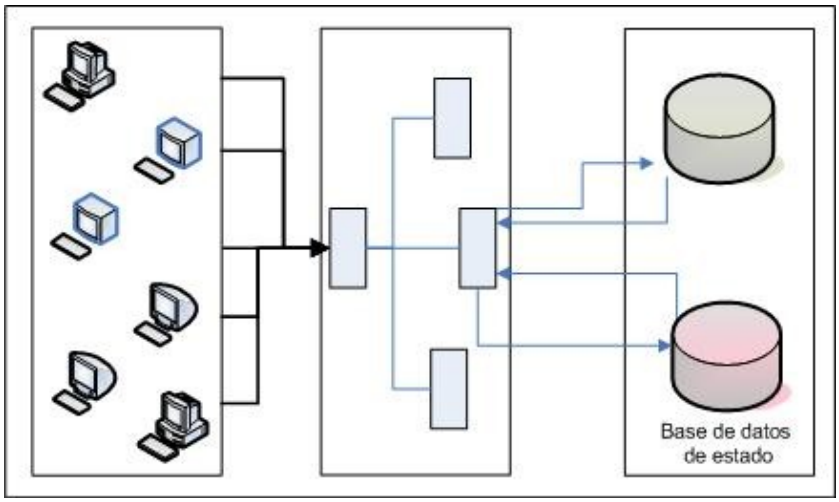


FIGURA 3.12 – Extensiones a las tradicionales arquitecturas distribuidas

Usualmente, y para evitar accesos remotos, estas bases de datos están ubicadas físicamente en el mismo servidor de aplicaciones que los componentes, y son del tipo *in-memory*[24] para evitar los riesgos del acceso a disco.

Otra opción en la administración de estado es el uso de un *middleware,* que se constituye como parte fundamental, con estado y autosuficiente de la infraestructura de una organización, que puede ser utilizada por otras soluciones.

G. Los Servicios deben poder ser descubiertos

Desde una perspectiva arquitectónica es deseable que los elementos que ofrecen lógica de aplicación sean fácilmente localizables.

Este proceso de buscar y encontrar lógica de aplicación dentro de un ambiente específico, se denomina *discovery.*

Un aspecto central de tal proceso es que es posible no saber de la existencia de la lógica de aplicación necesitada, antes de descubrirla.

---

24 **Base de datos in-memory:** http://en.wikipedia.org/wiki/In-memory_database

Descubrir que existe lo que se necesitaba implica evitar generar lógica de aplicación redundante. Descubrir que no existe aquello buscado implica que es posible definir el alcance del desarrollo

Este principio usualmente es clasificado como una extensión de infraestructura y por lo tanto asociado a arquitecturas empresariales.

Para que algo pueda ser descubierto dentro de una organización, necesita ser provisto de meta información que permita ser encontrado en búsquedas que estén dentro de su alcance.

La información necesaria para que un servicio sea descubrible es una combinación del contenido en el contrato de servicio y de cierta meta información

El proceso y tecnologías de descubrimiento mayormente están orientadas a humanos.

Los arquitectos o desarrolladores de la organización que necesiten buscar o construir nuevas piezas de software serán quienes mayormente necesiten de los medios para hacerlo (FIGURA 3.13)

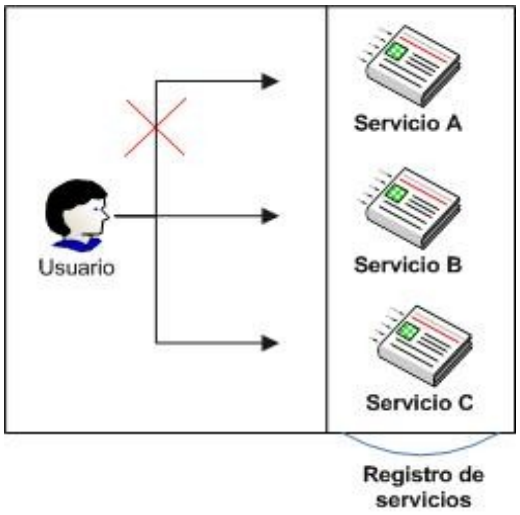


FIGURA 3.13 – Uso del registro de servicios

El uso de un registro de servicios para la implementación de este principio establece un mecanismo formal para facilitar la búsqueda y recuperación de meta información asociada a los servicios (FIGURA 3.14)

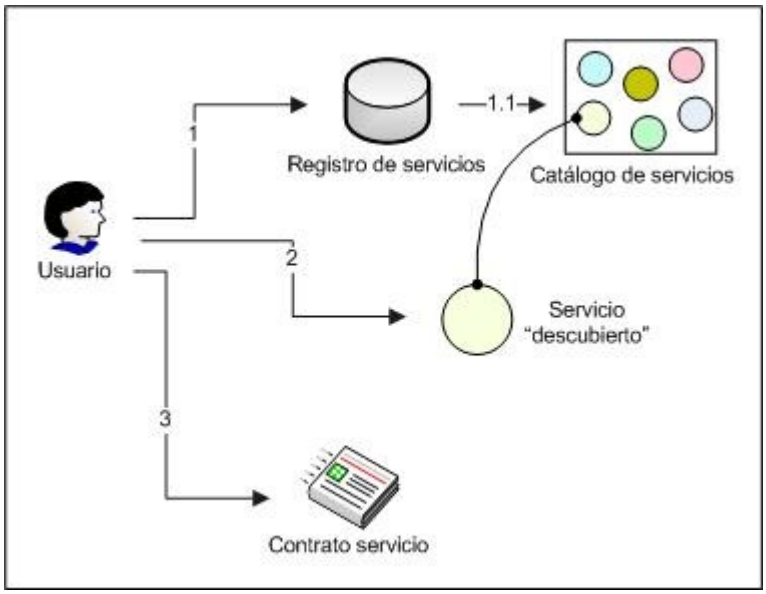


FIGURA 3.14 – Recuperación de información asociada al servicio descubierto

G.1. Tecnologías

Varias estrategias existieron para implementar el catálogo disponible de servicios, ya sean públicos o privados. Planillas de cálculo, directorios LDAP, o simplemente a partir de páginas web accesibles desde la intra/internet.

La llegada de la plataforma de servicios web formalizó y consolidó el estándar UDDI (*Universal Description, Discovery and integration*) para la realización del proceso de descubrimiento de servicios. Aunque no ha sido ampliamente adoptado, es un elemento destacable en el ámbito de la computación orientada a servicios.

3.2. Evolución de SOA

3.2.1. Cambios en las arquitecturas de software

La FIGURA 3.15 muestra como de las primeras arquitecturas monolíticas, donde las aplicaciones ejecutaban aisladas en las computadoras, se llega a las actuales arquitecturas orientadas a servicios, que facilitan la integración de aplicaciones heterogéneas al ocultar los detalles de la plataforma y tecnologías utilizadas para la implementación de los servicios. [9][12][16][38]

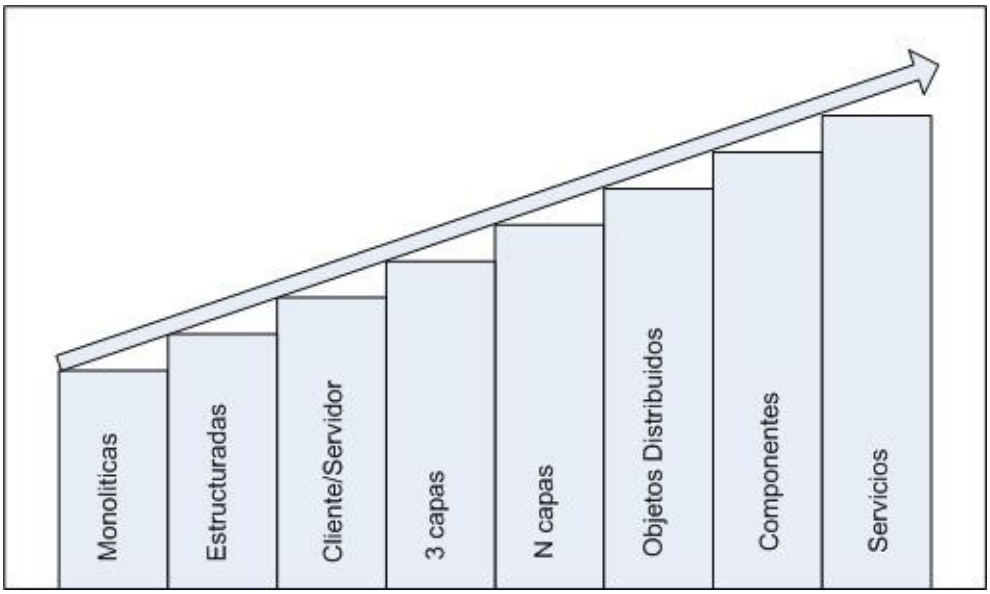


FIGURA 3.15 – Evolución de las arquitecturas de software (adaptada de [9])

En esta **evolución** de las arquitecturas IT tuvo influencia la necesidad de las organizaciones de colaborar con socios de negocios a través de sus sistemas IT.

En los 80's las aplicaciones eran mayormente verticales, cubriendo todas las necesidades de algún rubro en particular.

A principios de los 90's surgieron sistemas IT que le permitieron a las organizaciones crecer horizontalmente cooperando con socios de negocios. De esta manera surge el concepto de colaboración B2B (*Business to Business*), a través de componentes distribuidos en sistemas IT de diferentes organizaciones.

En la actualidad ya no es suficiente únicamente la colaboración entre organizaciones socias de negocios, sino que éstas también deben brindar acceso a sus servicios de negocios a sus clientes y/o empleados.

Surge el concepto de B2C (*Business to Consumer*) donde las aplicaciones clientes tienen acceso directo a los servicios electrónicos de negocios ofrecidos por las organizaciones.

Estos servicios deben ocultar la complejidad de la lógica interna de la organización, y ofrecer interfaces lo más abstractas posibles para abarcar diferentes escenarios de interacción. De esta manera, una aplicación cliente consume el servicio a través de interfaces bien definidas sin conocer los detalles internos de implementación. Se consolida el concepto de **arquitectura orientada a servicio** (SOA).

3.2.2. Arquitectura

Puede ser definida como la representación de un grupo de relaciones entre diferentes componentes de una solución de software compleja. Sirve como la

plantilla para el diseño global de la solución (FIGURA 3.16). [30][31]

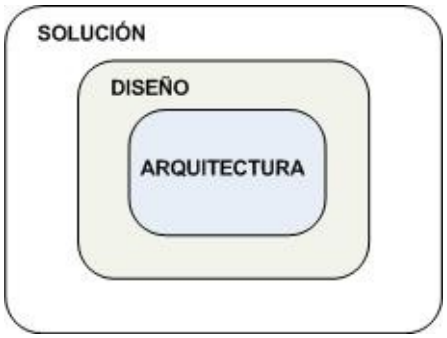


FIGURA 3.16 - Arquitectura

La arquitectura de una solución informática es importante, ya que sin ella no se podría, o sería sumamente dificultoso:

- Descomponer los requerimientos en piezas más pequeñas
- Obtener soluciones de calidad
- Administrar los cambios
- Obtener soluciones extensibles o reusables

3.2.2.1. Arquitectura de aplicaciones

En niveles bajos de granularidad, todo sistema posee aplicaciones ejecutándose para cumplimentar los objetivos de negocio de la organización. Estas aplicaciones son desarrolladas utilizando diferentes arquitecturas.

La arquitectura de las aplicaciones pueden ser consideradas como la representación de la estructura de los componentes e interacción entre ellos en el sistema [16]

La FIGURA 3.17 muestra una arquitectura de aplicación web típica, de 3 capas: presentación, negocios y datos

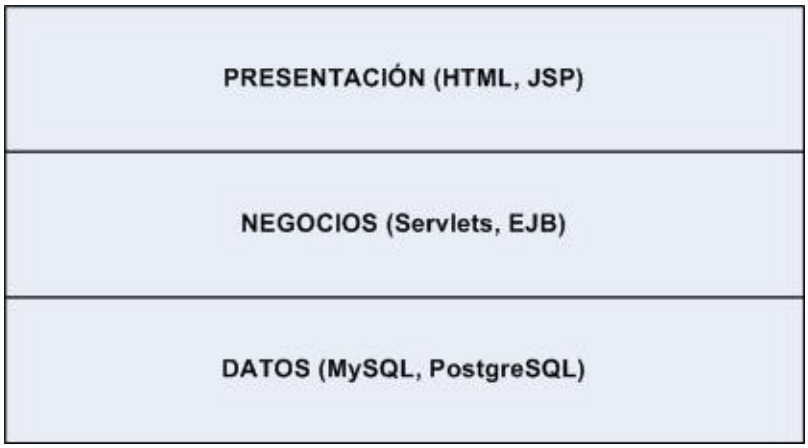


FIGURA 3.17 - Arquitectura de Aplicación

Cada organización usualmente tiene múltiples aplicaciones, con diferentes arquitecturas, que solucionan distintas necesidades de negocio.

A. Arquitecturas Cliente/Servidor

Teniendo en cuenta que los ambientes donde una pieza de software que requiere o recibe información desde cualquier otra aplicación pueden ser considerados cliente/servidor, prácticamente cualquier variación de arquitecturas de aplicación tienen un elemento de interacción cliente-servidor en ellas.

Sin embargo, cuando en este trabajo se hace referencia al término cliente/servidor es en base a la definición aceptada por la industria que refiere, usualmente, a una generación particular de ambientes en los cuales los clientes y servidor juegan roles distintos con características de implementación diferentes.

Es así, entonces, que las arquitecturas cliente/servidor separan el cliente, quien siempre inicia el pedido, del servidor, quien recibe, procesa y responde a ese pedido.

Este tipo de arquitecturas construidas básicamente en 3 capas, pueden ser divididas en arquitecturas de 1, 2, 3 o N niveles. Como fue nombrado anteriormente, usualmente las capas se clasifican en: presentación, negocios y datos.

- **Capa presentación**: es la capa con la cual interactúa el cliente, que permite iniciar el proceso de negocios.
- **Capa negocios**: procesa la información recibida de la capa de presentación para realizar los objetivos de negocios definidos en base a ciertas reglas de negocios establecidas.
- **Capa de datos:** almacena los datos, y eventualmente la lógica, necesaria para cumplimentar los objetivos de negocios.

B. Arquitecturas distribuidas en Internet

En respuesta a los altos costos y limitaciones asociadas con las arquitecturas cliente/servidor de 2 capas es que surge el concepto de aplicaciones basadas en componentes.

Aparecen las arquitecturas cliente/servidor multi-capa que dividen el antiguo ejecutable monolítico del cliente en componentes con capacidades para procesar múltiples pedidos concurrentes.

Adicionalmente, se reemplazan las antiguas conexiones cliente/servidor con las bases de datos, por invocaciones (cliente/servidor) a procedimientos remotos (RPC).

Las tecnologías RPC (CORBA, DCOM) permitían comunicación remota entre componentes residentes en la parte cliente, y servidores. Se complejiza el mantenimiento y administración de la arquitectura al agregarse una nueva capa de middleware –servidores de aplicaciones, monitores de transacciones- para brindar soporte a este nuevo tipo de interacciones.

Con el arribo de la WWW a mediados de los 90's como medio viable de comunicación para las tecnologías de computación, los ambientes cliente/servidor multi-capa comenzaron a incorporar tecnología de internet.

El cambio más significativo fue el reemplazo de los componentes de software en el cliente, por el navegador web. A pesar de limitar las capacidades gráficas en los clientes, el principal cambio fue haber desplazado prácticamente toda la lógica de negocios del lado del servidor.

Las arquitecturas distribuidas de internet introdujeron una nueva capa: el servidor web. Básicamente, esta nueva capa fue necesaria al momento de reemplazar los antiguos protocolos privativos RPC, por el estándar HTTP para realizar las comunicaciones entre clientes y servidor.

3.3. Definiendo SOA

Es difícil elegir una definición de SOA, principalmente porque existen muchas, y algunas difieren bastante entre sí. [12][13]

Según la definición dada por Wikipedia[25], SOA es considerado

> *“un concepto de arquitectura de software que define la utilización de servicios para dar soporte a los requisitos del negocio. Permite la creación de sistemas altamente escalables que reflejan el negocio de la organización, a su vez brinda una forma estándar de exposición e invocación de servicios (comúnmente pero no*

25 **SOA:** http://es.wikipedia.org/wiki/SOA (a Octubre de 2008)

> *exclusivamente servicios web), lo cual facilita la interacción entre diferentes sistemas propios o de terceros"*

A. SOA es un paradigma

SOA no es en sí mismo una arquitectura concreta, sino *algo* que conduce a una arquitectura concreta. Ese *algo* puede ser llamado un mecanismo, paradigma, estilo, concepto, o representación. Esto es, SOA **no** es una herramienta o framework que puede adquirirse.

Es una manera de pensar que conducen a decisiones al momento de diseñar una arquitectura de software concreta.

B. SOA promueve la reusabilidad y débil acoplamiento

SOA describe una arquitectura de sistemas IT basados en el principio de producción de servicios de negocios reusables, ensamblados en componentes de software de manera tal que los proveedores y consumidores de los servicios de negocios/infraestructura sean débilmente **acoplados**.

Específicamente con SOA, los proveedores y consumidores no necesitan tener conocimiento de las tecnologías, plataformas, ubicación, sistemas operativos, lenguajes de programación en que fueron construidos los servicios o particularidades de la elección de los ambientes de cada uno.

SOA promueve el débil acoplamiento entre los componentes, de manera de facilitar la reusabilidad

El débil acoplamiento entre servicios surge de la necesidad de las aplicaciones de ser más ágiles, y adaptables a los cambios. Intentos anteriores en estrategias de integración no condujeron a soluciones abiertas e interoperables ya que principalmente se basaban en interfaces de programación de aplicaciones (APIs) propietarias, que requerían gran esfuerzo de coordinación entre los distintos componentes participantes.

Esta necesidad de agilidad y adaptabilidad ante cambios facilitó el crecimiento y adopción de SOA, concepto acuñado a mediados de los 90's, pero del que recientemente han surgido implementaciones prácticas reales y concretas.

Esta realidad ha permitido la creación y adaptación de soluciones empresariales en ambientes de negocios en constante cambio. Los desarrolladores pueden aprovechar el potencial que se adquiere al reutilizar componentes IT existentes (aplicaciones heredadas, *middleware*, plataformas móviles, diferentes base de datos y demás) que facilita el rápido ensamble de aplicaciones, con el consecuente beneficio de reducción de costos, tiempos, y mantenibilidad posterior.

C. Integración de sistemas y SOA

Desde la perspectiva de integración de sistemas, SOA es considerado como otra forma de conectar aplicaciones a través de una red mediante protocolos de comunicación conocidos.

Se lo define como un estilo arquitectónico para la construcción de aplicaciones de software integradas que utilizan servicios disponibles en una red (inter/intranet).

De esta manera, permitiría a los desarrolladores considerar las aplicaciones como servicios de red que pueden ser combinados con el objetivo de crear procesos de negocios más complejos, de forma más rápida y fácil que utilizando otros métodos similares.

Considerando las características y aplicaciones de SOA en los actuales complejos y heterogéneos ambientes de computación, puede facilitar que las diferentes organizaciones puedan operar más eficientemente y tengan mayor capacidad de adaptación a los cambios que se van sucediendo, promoviendo y adoptando, de esta manera, el concepto de SaaS (*Software as a Service*)

3.4. Conceptos de SOA

SOA se basa fuertemente en algunos conceptos técnicos con las características anteriormente descriptas: servicios, interoperabilidad, débil acoplamiento, neutralidad de la plataforma, reusabilidad e integración.

A. Servicios

Un proceso de negocios (o, en general, cualquier proceso) es un conjunto organizado de tareas. Ejecutar ese proceso es una cuestión de coordinar la ejecución de dichas tareas.

La idea central de SOA es tener componentes especializados que realicen tales tareas; esos componentes son los llamados **servicios**.

El concepto de servicio es quizás el más importante en SOA.

Un servicio SOA puede ser un objeto simple, un objeto complejo, una colección de objetos, procesos conteniendo varios objetos, procesos conteniendo otros procesos, e incluso una colección de aplicaciones con una única salida.

Desde el exterior el servicio es visto como una única entidad, pero dentro de él puede tener el nivel de complejidad necesario.

Una **definición** simple de **servicio** es la que lo considera:

> *"una unidad de funcionalidad que un proveedor de servicio (service provider) deja disponible en el ambiente (internet, LAN) a través de una interface, para que los consumidores del servicio (service consumers) pueden hacer uso"*

Esencialmente, un servicio está compuesto de su interface, implementación y mecanismos de invocación (FIGURA 3.18).

La implementación del servicio está determinada por la funcionalidad que el servicio pretende ofrecer. La interface permite "describir" el servicio, siendo la mejor manera de hacerlo a través de XML, el cual facilita el débil acoplamiento entre los clientes y la implementación del servicio.

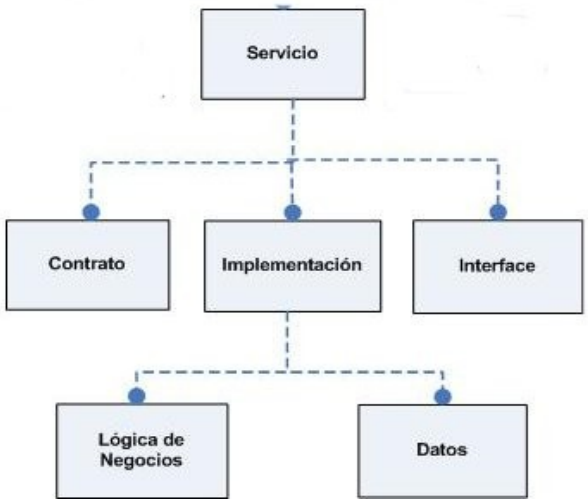


FIGURA 3.18 – Componentes típicos de un servicio

WSDL, basado en XML, es un mecanismo globalmente aceptado para describir la interface de servicio. La interface del servicio describe el nombre del servicio, operaciones en el servicio, entradas, salidas y mensajes fallados. No incluye ninguna información relacionada con los mecanismos de invocación.

Esta información es incluida en los *bindings* del servicio, usualmente en el archivo descriptor del servicio.

En particular, WSDL soporta la inclusión de información de *bindings*, la cual detalla que protocolo es utilizado y exactamente cómo serán enviadas las entradas y obtenidas las salidas.

Los principales atributos de un servicio son:

- Es definido con el **nivel correcto de granularidad** desde el punto de vista del consumidor.
- Es **autodescriptivo**. Los potenciales consumidores pueden aprender por si

mismos como invocar un servicio

- Es tecnológicamente **agnóstico.** Los potenciales consumidores no tienen restricciones en cuanto a plataformas de hardware/software para la invocación del servicio. Idealmente un servicio puede interoperar con cualquier consumidor.
- Es **descubrible**. Los consumidores que buscan por determinado servicio pueden encontrarlo buscando en repositorios públicos o privados, denominados registro de servicios
- Un servicio puede ser **compuesto con otros servicios** para formar servicios de más alto nivel o mayor granularidad
- Los servicios **no tienen estado**, facilitando el débil acoplamiento entre proveedores y consumidores.

A.1. ¿Cómo encapsulan la lógica?

Para lograr su independencia, los servicios encapsulan su lógica de negocios dentro de determinado contexto. Este contexto puede ser una simple tarea de negocios, una entidad de negocios, o bien cierto agrupamiento lógico.

El tamaño y alcance de la lógica que encapsula el servicio puede ser variable, pudiendo componer varios servicios para formar servicios de mayor granularidad.

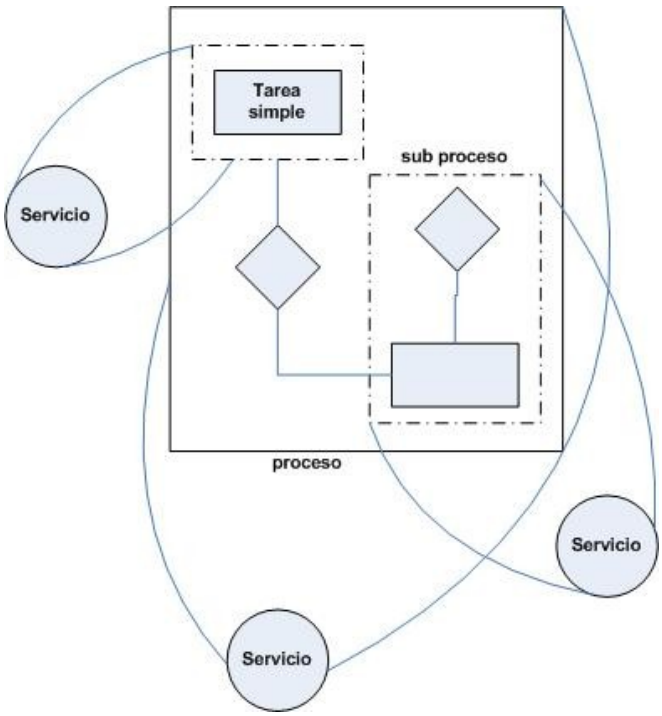


FIGURA 3.19 – Lógica encapsulada por un servicio

La FIGURA 3.19 muestra como un servicio puede encapsular una tarea realizada

por un paso individual, un subproceso que comprende un conjunto de pasos, e inclusive la lógica de un proceso completo.

A.2. ¿Cómo se relacionan?

Dentro de SOA, los servicios puede ser usados por otros servicios o aplicaciones.

Sin importar como está basada la relación entre servicios, para interactuar éstos deben tener conocimiento el uno del otro. Este conocimiento es logrado a partir del uso de descriptores de servicios (FIGURA 3.20)

La descripción de un servicio en su forma más básica establece un nombre del servicio, los datos que espera y aquellos que retornará. La manera en la cual los servicios utilizan los descriptores de servicios resulta en relaciones clasificadas como débilmente acopladas.

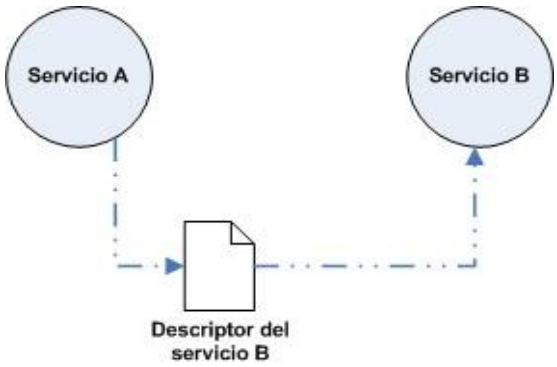


FIGURA 3.20 – Relación entre servicios mediante descriptores

Para que la interacción entre servicios tenga significado, se espera que éstos intercambien información. Un framework de comunicación que permita preservar la relación de débil acoplamiento es requerido. Uno de tales frameworks es la mensajería.

A.3. ¿ Cómo se comunican ?

Luego que un servicio envía un mensaje, pierde el control de lo que pasa con ese mensaje. Es por ésto que se requiere que los mensajes existan como unidades de comunicación independientes; es decir, que, al igual que los servicios, los mensajes deberían ser autónomos.

Para cumplimentar este requisito los mensajes pueden ser provistos de la suficiente inteligencia para autogobernar las partes de lógica de procesamiento (FIGURA 3.21)

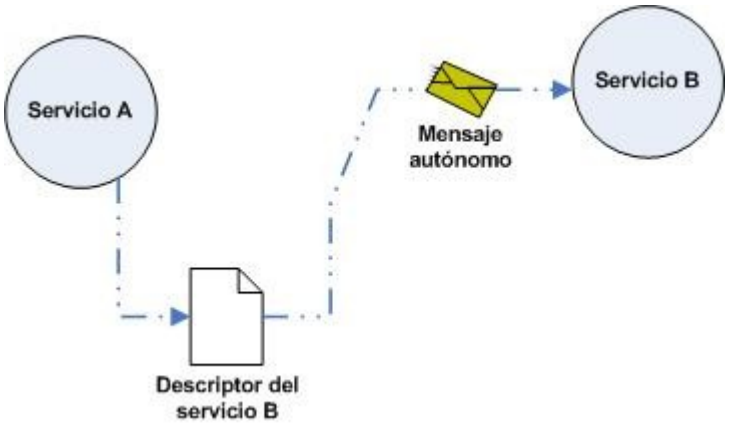


FIGURA 3.21 – Comunicación entre servicios

Los servicios que proveen descriptores y se comunican vía mensajes forman una arquitectura básica. Esta arquitectura, similar a las antiguas arquitecturas distribuidas que soportan mensajería y separación de interfaces de la lógica de procesamiento, se distingue por como sus tres componentes principales (servicios, descriptores, mensajes) son diseñados. La clave es la “orientación a servicios”

A.4. ¿Cómo son diseñados?

Como ocurrió con la orientación a objetos, la orientación a servicios se ha convertido en una aproximación diferente al diseño de aplicaciones que introduce principios comúnmente aceptados que gobiernan el posicionamiento, diseño y construcción de componentes arquitectónicos.

Cuando una solución está comprendida de unidades de lógica de procesamiento orientadas a servicios, se convierte en lo que se denomina una solución orientada a servicios.

Algunos de los principios de la orientación a servicios fueron analizados en secciones anteriores, mientras que otros serán profundizados a medida que se avance en el desarrollo de este capítulo; igualmente los principales serán enumerados en esta sección:

Algunos aspectos claves de estos principios de diseño son:

- **Débil acoplamiento**: los servicios mantienen una relación que tiende a minimizar las dependencias entre sí
- **Autonomía:** los servicios tienen el control sobre la lógica que encapsulan.
- **Abstracción:** más allá de lo descripto en el contrato de servicio, los servicios ocultan los detalles de su lógica al mundo exterior
- **Reusabilidad**: la lógica es dividida en servicios con el objetivo de promover

la reusabilidad

- **Sin estado:** los servicios tienden a minimizar el estado que mantienen para las actividades que realizan.
- **Contrato de servicio:** los servicios adhieren a un acuerdo de comunicación, definido a partir de contratos de servicios y documentos relacionados.

Conociendo los componentes que comprenden la arquitectura, y un conjunto de principios de diseño que permitan estandarizar tales componentes, solamente resta la plataforma de implementación que permita agrupar todas estas piezas, y de esta manera construir soluciones orientadas a servicios. ESB y servicios web son las tecnologías candidatas para tal plataforma.

A.5. ¿ Cómo son construidos ?

Como ha sido mencionado en las secciones anteriores, el término “orientado a servicios” ha existido incluso antes que la llegada de los servicios web y ESB. Sin embargo, aún ningún avance tecnológico ha sido el más adecuado para la construcción de SOA, como lo son la combinación de servicios web y ESB.

B. Elementos de los servicios

La FIGURA 3.22 muestra los elementos de un servicio típico

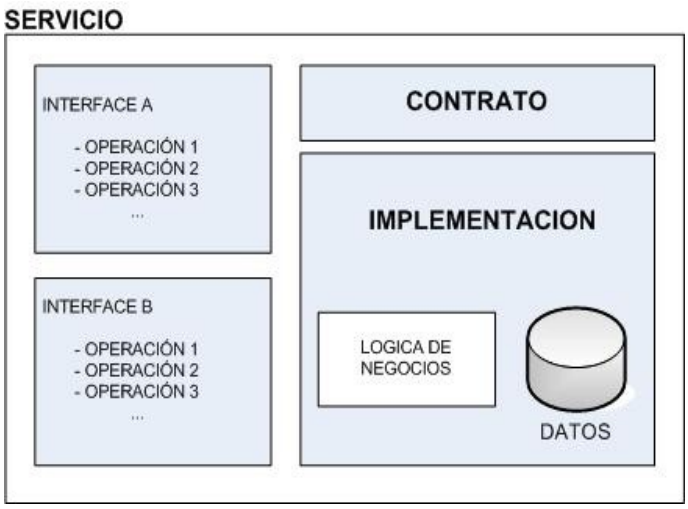


FIGURA 3.22 – Elementos principales de un servicio

B.1. Contrato

El contrato contiene una especificación del propósito, funcionalidad, restricciones

y uso del servicio.

Generalmente el contrato también contiene la definición de interface. Es recomendable el uso de lenguajes formales, tales como IDL o WSDL, para la definición de tal interface, lo que permitiría abstraer las tecnologías, logrando una arquitectura más débilmente acoplada.

B.2. Interface

La funcionalidad provista por el servicio es entregada al consumidor del servicio a través del uso de las interfaces.

La definición o descripción de la interface contiene información referente a los argumentos que se esperan recibir, funcionalidad provista y eventualmente la calidad del servicio.

La interface recibe los pedidos entrantes y los delega a la lógica de negocios encargada de procesarlos.

B.3. Implementación

En esta parte reside la funcionalidad del servicio. Es la realización técnica de lo especificado en el contrato.

Básicamente consiste de la lógica de negocios que representa la lógica de programación dentro del servicio, de los datos que el servicio se encarga de recuperar, persistir y eventualmente del manejo de mecanismos de bloqueo y transacciones

SOA divide la estructura de una arquitectura en componentes pequeños: servicios. Estos servicios son los bloques flexibles sobre los cuales se apoya la arquitectura.

La organización de estos servicios se realiza a partir de la clasificación en tipos de servicios de acuerdo a su granularidad o nivel de abstracción.

Aunque los servicios pueden tener cualquier funcionalidad, el desafío consiste en definir interfaces de servicios con el nivel de abstracción correcto. Los servicios deberían proveer funcionalidad con granularidad gruesa.

B.4. Repositorio o Registro de Servicios

Luego de definir los servicios, el próximo paso (opcional) es publicarlos en un registro o repositorio.

SOA usa el paradigma "*find, bind and invoke*". En este paradigma, los proveedores de servicios registran sus servicios en el registro. Este registro es usado por consumidores para encontrar (*find*) los servicios que concuerden con determinados criterios de búsqueda. Si el registro tiene tal servicio, provee (*bind*)

al consumidor del contrato y dirección lógica (*invoke*) para tal servicio.

UDDI es la especificación que permite definir un registro de servicios para Servicios Web.

Almacena información sobre los proveedores de servicios, implementaciones y metadata de los servicios. Los proveedores de servicios utilizan UDDI para publicar los servicios que ofrecen.

Los servicios consumidores pueden utilizarla para la búsqueda de los servicios que concuerden con sus necesidades, y obtener la metadata necesaria para utilizar tales servicios.

De esta manera UDDI ofrece una plataforma estándar e interoperable que facilita a las aplicaciones y servicios a encontrar y usar fácil y dinámicamente servicios web a través de la red.

C. Interoperabilidad

En ambientes heterogéneos uno de los principales objetivos es poder conectar fácilmente los sistemas. Esta idea no es nueva. Antes de SOA ya existía el concepto de EAI (*Enterprise Application Integration*). Sin embargo SOA facilita la interoperabilidad entre sistemas al implementar lógica de negocios encapsulada (servicios) que se distribuye en múltiples sistemas distribuidos.

D. Débil acoplamiento

Edsger Dijkstra[26] afirmaba que "***mínimo acoplamiento, máxima cohesión***" indicaba el desarrollo de aplicaciones de alta calidad.

Tradicionalmente las aplicaciones han sido sumamente rígidas tanto que, una vez construidas, eran difícilmente adaptables. Para evitar estas situaciones, era necesario minimizar el número de dependencias.

Una de las principales características de SOA es que facilita el desarrollo y despliegue de módulos de software débilmente **acoplados**. Estos módulos o servicios pueden operar y cooperar de manera distribuida e independiente, sin ningún requerimiento en cuanto a lenguajes de programación, plataformas de hardware/software, ubicación física, conocimiento de modelos de objetos, sistemas operativos y estructuras de base de datos.

Estos servicios son independientes e interoperables.

Gráficamente, la FIGURA 3.23 muestra las diferencias entre arquitecturas alta y débilmente acopladas

26 **Edsger Dijkstra**: es.wikipedia.org/wiki/Dijkstra

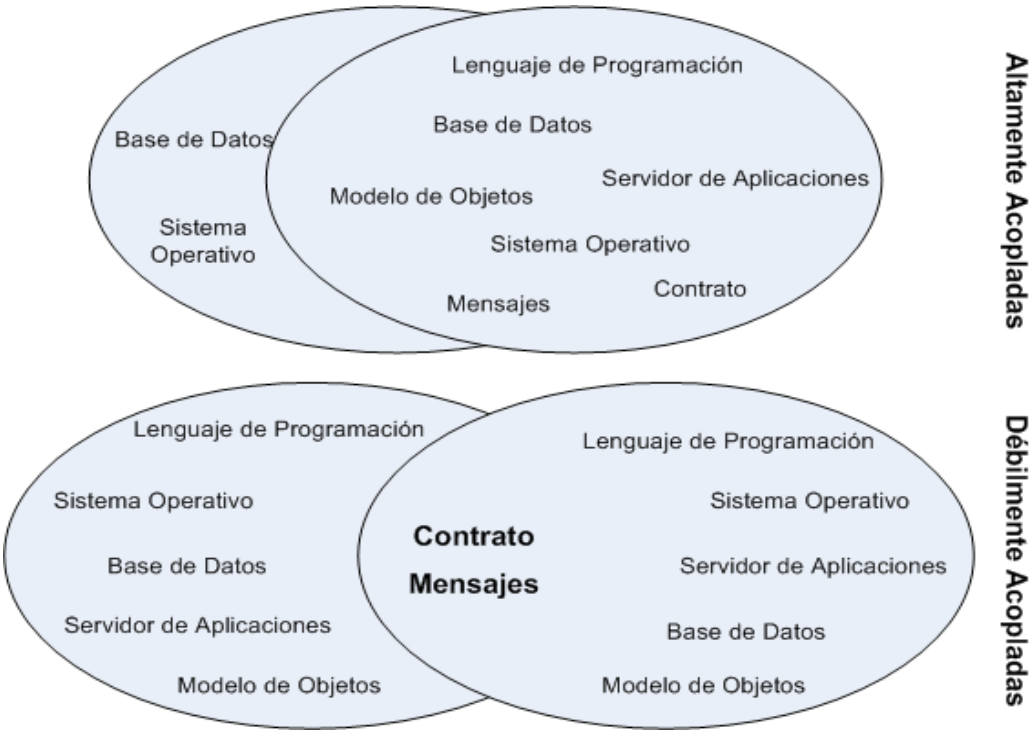


FIGURA 3.23 - Arquitecturas débilmente acopladas vs altamente acopladas

D.1. Arquitecturas débilmente acopladas

Algunos aspectos del débil acoplamiento son fácilmente implementables a partir del uso de servicios basados en mensajes con granularidad gruesa que se comuniquen de manera asincrónica.

En general, es siempre recomendable el uso de servicios basados en mensajes, y es un concepto que esta adquiriendo mayor popularidad.

Al principio, las primeras implementaciones de servicios web usaban codificación *RPC/SOAP*; sin embargo, la tendencia actual se inclina por el uso de servicios del estilo *Document/Literal,* es decir, servicios basados en mensajes.

La otra tendencia hacia la adopción de servicios asincrónicos esta relacionada con el movimiento desde RPC (*Remote Procedure Call*) a mensajería. La mayoría de los servicios RPC son sincrónicos, siendo las principales tecnologías las nombradas CORBA, RMI, EJB, COM/DCOM.

Un cambio hacia servicios basados en mensajería implica un cambio en la granularidad y manera en que los sistemas son diseñados.

Fundamentalmente, una arquitectura débilmente acoplada tiene las siguientes características:

- Robusta

- Mantenible
- Facilita la mensajería asincrónica
- Escalable
- Independiente de la plataforma, ubicación, tiempo
- Adaptable

El débil acoplamiento del servicio y sus consumidores es sumamente importante en plataformas SOA. La invocación del servicio debe ser totalmente independiente de la implementación del mismo. La única dependencia puede ser el descriptor del servicio (WSDL por ejemplo) y posiblemente el protocolo de comunicación (SOAP por ejemplo).

E. Neutralidad de la plataforma

El uso de mensajes basados en XML promueve la neutralidad tecnológica de la plataforma, al ser un formato de representación de datos estándar.

F. Reusabilidad

Al reutilizar componentes (FIGURA 3.24) el desarrollador usa librerías y código ya compilado que debería ejecutar de acuerdo a lo previsto. En cuanto a servicios, por su parte, significa reutilizar código ejecutándose, presumiblemente ya sirviendo a otros servicios que requieran su funcionalidad.

El nivel de reutilización es uno de los principales indicadores del éxito de una arquitectura orientada a servicios. Mayor es el número de procesos que reutilizan un servicio, más fácil es su mantenimiento y pruebas de su código.

Los servicios deben ser lo suficientemente independientes de requerimientos de las aplicaciones, y debe ser fácilmente extensible a través de parametrizaciones, o descomposición en servicios de granularidad más fina.

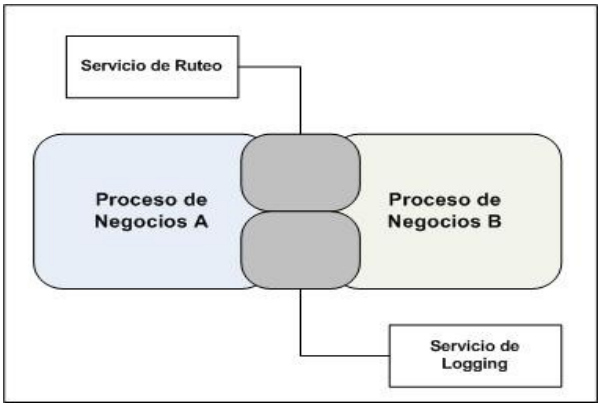


FIGURA 3.24 – Reusabilidad de servicios

G. Integración

SOA intrínsecamente es una metodología de integración de aplicaciones, al promover el diseño y codificación de manera modular y compartible.

La integración puede ocurrir en diferentes niveles, incluso internamente dentro de una misma organización, permitiendo la reutilización de servicios en diferentes aplicaciones de uso privado. Sin embargo, la opción de exponer públicamente servicios que puedan ser accedidos por aplicaciones u otros servicios externos, establece las bases de una nueva forma de construir, integrar y componer aplicaciones heterogéneas.

Internet se está convirtiendo en el contenedor de servicios de negocios que pueden ser accedidos, utilizados y ensamblados para crear nuevos procesos de negocios, con el objetivo de facilitar el intercambio B2B (*business to business*) y B2C (*business to consumer*) [9]

3.5. Elementos de SOA

En la sección anterior se enumeraron los conceptos técnicos requeridos para la construcción exitosa de una arquitectura orientada a servicios. Como ha sido remarcado en varias ocasiones, una SOA no puede ser comprada, sino que debe ser construida y diseñada con elementos concretos.

Entre tales elementos es necesario destacar aquellos más importantes y requeridos al momento de plantearse el diseño de SOA [17]

A. Infraestructura

La infraestructura es la parte técnica de SOA que facilita la alta interoperabilidad. El principal componente de la infraestructura es el ESB (*Enterprise Service Bus*), término surgido de EAI, donde se lo denominaba EAI bus o *Enterprise Bus*.

La clave del ESB es que permite la interacción de servicios en ambientes heterogéneos, ofreciendo ruteo inteligente, transformación de datos, *logging*, seguridad, transaccionalidad, entre otras características fundamentales. En el capítulo 4 se profundizará más sobre las características y propósito de los ESBs.

B. Arquitectura

La arquitectura es necesaria para restringir todas las posibles opciones y combinaciones para la construcción de SOA, que conduzcan a sistemas productivos y mantenibles.

Este elemento es necesario para organizar los diferentes tipos de servicios, definir la “cantidad” de débil acoplamiento entre los sistemas, las políticas, reglas, prácticas e infraestructura tecnológica que formarán parte integral de la

arquitectura orientada a servicios global.

C. Gobierno de SOA

Básicamente es la estrategia global que permite organizar los procesos involucrados en la construcción de SOA. Este elemento incluye a las personas encargadas de combinar los distintos conceptos SOA para lograr un resultado que funcione y sea apropiado; lograr, por ejemplo, que los aspectos de infraestructura tecnológica, arquitectura y procesos convivan, y coordinen adecuadamente.

3.6. SOA y su relación con otras tecnologías

A. Relación con Servicios Web

Algunas de las definiciones de SOA incluyen el término servicios web. Sin embargo, como se ha remarcado previamente **SOA no es lo mismo que servicios web.**

SOA es un paradigma, mientras que los servicios web son una posible manera de construir la infraestructura utilizando una estrategia de implementación específica.

Los servicios web se han establecido como la tecnología más aceptada para las implementaciones de SOA. Sus características tecnológicas, y el hecho de estar basado en estándares, lo hacen coherente con los principios propuesto por el paradigma SOA, que pueden ayudar a establecer la infraestructura básica requerida.

Es necesario remarcar que existen otras alternativas de implementación de SOA, que han estado más tiempo en el mercado, que están más consolidadas, pero en general, no son tan flexibles como los servicios web para ser adoptadas [38]

B. Relación con Objetos Distribuidos

Existen múltiples opciones para el diseño, implementación de sistemas distribuidos. Una de tales aproximaciones es CORBA, que utiliza el concepto de objetos distribuidos.

Básicamente la idea es permitir el acceso remoto a sistemas externos, posibilitando la invocación remota a métodos de tales objetos.

Este tipo de interfaces a sistemas de granularidad muy fina, condujo a la idea de tener un objeto de negocios general que abarcara todo el sistema distribuido.

La noción de granularidad gruesa tiene cierta similitud con el patrón de diseño

Façade[27], popular en plataformas JEE y .NET. Por ejemplo, en aplicaciones JEE que utilicen tecnología EJB para encapsular la lógica de negocios, este patrón es útil para abstraer la funcionalidad que está relacionada y es accedida desde diferentes *front-ends* (JSP/Servlets, GUI, J2ME) en objetos "*façade*" que exporten operaciones de granularidad gruesa de la lógica de negocios del sistema.

Sin embargo, en la práctica esta solución no es escalable. Tener objetos *façade* para el acceso a los sistemas conectados introduce mucha centralización, muchas dependencias y en general complejiza la organización.

SOA promueve el concepto contrario a los objetos distribuidos. Con SOA los datos son intercambiados entre diferentes sistemas, y cada sistema opera sobre su copia local de los mismos. Esta aproximación desacopla los sistemas entre sí, y facilita su escalabilidad [32]

3.7. Tecnologías y estándares fundacionales de SOA

SOA es considerado el paradigma que ha evolucionado de las tradicionales arquitecturas cliente/servidor. En los sistemas cliente/servidor la interface de usuario, lógica de negocios y funciones de administración de los datos están separadas de tal manera que cada una puede ser implementada usando plataformas y tecnologías que mejor se adapten a esas tareas [9]

Con SOA estas funciones (específicamente lógica de negocios/aplicación) se descomponen en niveles más finos de granularidad.

Por ejemplo, en vez de implementar lógica de negocios en módulos de software monolíticos, un sistema basado en SOA incorpora servicios ejecutando en diferentes plataformas de software, inclusive servicios alojados en proveedores de aplicación remotos.

La construcción de una aplicación SOA no sólo necesita identificar los servicios que la componen, sino también es necesario describir como será el flujo de trabajo global entre los distintos servicios que conforman la aplicación

SOA también es considerado como una evolución en paradigmas conocidos para el diseño de sistemas informáticos: diseño orientado a objetos y diseño de componentes distribuidos.

La orientación a objetos[28] es un diseño que facilita la construcción de sistemas extensibles a partir de la abstracción y ocultamiento de los módulos y partes complejas de un sistema.

Los componentes distribuidos, por su parte, son una extensión al mecanismo de comunicación RPC, donde ya no es un método *remoto* lo que se está invocando,

---

27 **Patrón de diseño Façade:** http://es.wikipedia.org/wiki/Facade_(patrón_de_diseño)

28 En el Anexo I se hace un análisis comparativo entre orientación a objetos y servicios

sino que se está accediendo a un *objeto remoto*, con sus datos y funcionalidad encapsulada en una sola entidad, al igual que los objetos en OOP. Ejemplos de componentes remotos son las tecnologías EJB y DCOM.

Los servicios basados en componentes son similares a los componentes distribuidos.

Sin embargo, los componentes distribuidos están fuertemente acoplados a los protocolos y tecnologías que lo implementan. Es así, por ejemplo, que las aplicaciones EJB necesitan que los clientes también sean implementados usando EJB.

Luego, son recomendables las tecnologías basadas en estándares abiertos para la implementación de servicios sin que dependan de proveedores o librerías particulares.

3.7.1. Internet y tecnologías "livianas"

Internet trajo consigo numerosos cambios y mejoras en cuestiones relacionadas con infraestructuras IT, y en particular, facilitó la rápida consolidación y adopción de SOA, al permitir el acceso remoto de servicios a través de internet y HTTP.

Otra tecnología "liviana" adoptada por SOA fue XML, que con el tiempo fue ganándose su lugar como el formato de intercambio de datos más aceptado por la industria a nivel global.

Estas tecnologías, HTTP y XML, junto al poder de internet y especificaciones adicionales (SOAP, WSDL, UDDI) facilitaron el desarrollo de un mecanismo simple para la invocación remota de funcionalidad: los servicios web

Las invocaciones, en la forma de documentos XML de acuerdo a las especificaciones SOAP y WSDL, son enviados a una dirección web, donde un componente del lado del servidor recibe el documento, lo interpreta y ejecuta el servicio requerido.

El desarrollador del servicio solamente necesita implementar por única vez la interface abstracta de acceso al servicio web, y el mecanismo de acceso a la funcionalidad de negocios del lado del servidor.

Una vez definido el servicio, la aplicación que necesita acceder a esa funcionalidad puede hacerlo a través del servicio web.

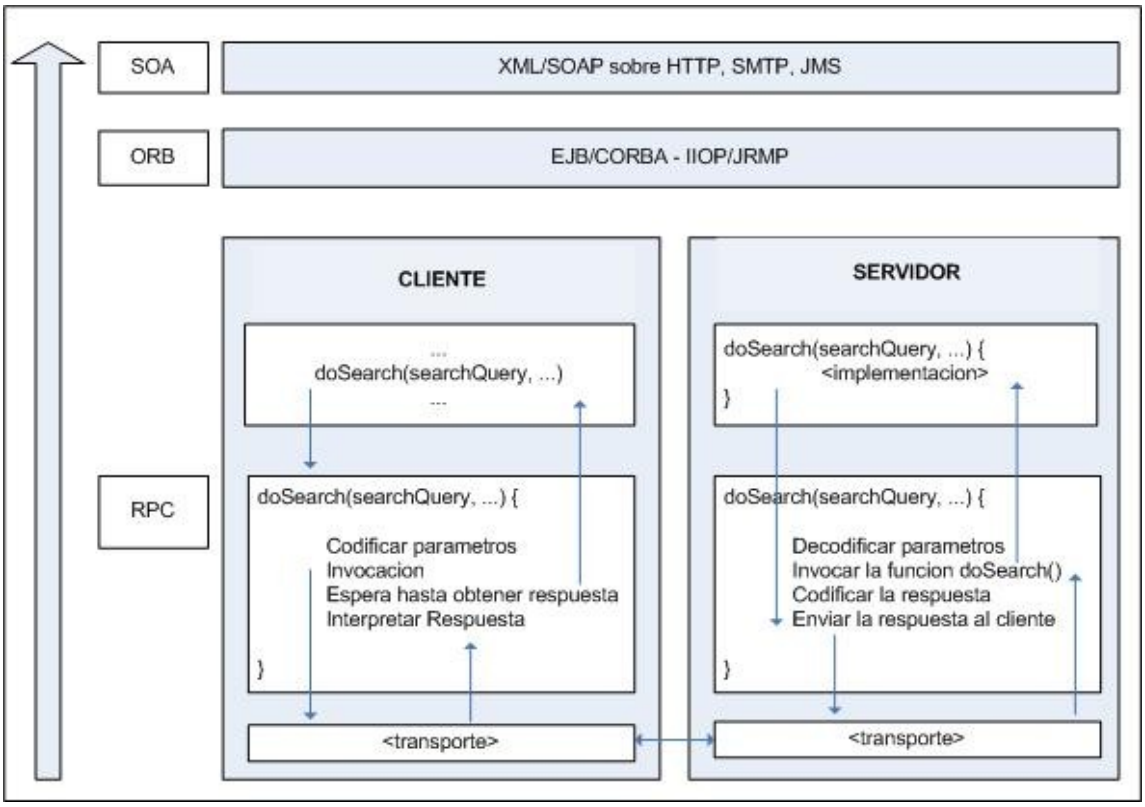


FIGURA 3.25 – Estándares y Tecnologías en SOA (adaptada de [9])

De esta manera, se obtiene el acceso a una aplicación remota de forma sencilla e independiente de las tecnologías e infraestructura utilizada del lado del servidor, y utilizando tecnologías estándares, abiertas y "livianas".

En el proceso de consolidación de SOA, el uso de estándares en general, y de la madurez de las tecnologías surgidas en la web (FIGURA 3.25) fueron claves para su crecimiento y adopción.

Específicamente era necesario tener una manera estándar de representar componentes de software. CORBA intento serlo, pero fue visto como dificultoso en su uso, y por lo tanto, no llego a ser un estándar reconocido universalmente por la industria IT. En los 90's Microsoft introduce DCOM (*Distributed Component Object Model*) como una alternativa, pero sin brindar soporte para CORBA.

A finales de los 90's surgió con mayor fuerza la posibilidad de utilizar estándares simples tales como HTML y HTTP para enlazar usuarios con las diferentes fuentes de información. Comienzan las investigaciones de como utilizar tales tecnologías fundacionales de Internet, para enlazar sistemas de computación entre sí. Son los primeros pasos de los servicios web.

SOA por si mismo es una construcción abstracta de como el software debería funcionar conjuntamente. Se basa fuertemente en tecnologías e ideas concretas, implementadas a partir de XML, Servicios Web, HTTP, entre otras. Adicionalmente requiere soporte de aspectos relacionados con la seguridad,

administración de políticas, mensajería confiable, transaccionalidad, entre otros, para que los sistemas puedan funcionar correctamente.

Por otra parte, los servicios web fueron diseñados para soportar interacciones interoperables máquina-máquina sobre una red. Esta interoperabilidad es obtenida a partir del uso de los estándares abiertos basados en XML antes descriptos: WSDL, SOAP y UDDI.

La WS-I[29] (Web Services Interoperability Organization) es una organización abierta que promueve la interoperabilidad entre los servicios web al proveer de herramientas y guías que permitan a los desarrolladores construir servicios web interoperables.

Los servicios web no son estrictamente necesarios para construir una SOA. CORBA, sistemas MOM y JMS también pueden ser utilizados para la construcción de arquitecturas orientadas a servicios.

Sin embargo las implementaciones contemporáneas tienden a utilizar protocolos y tecnologías estándares para efectivamente administrar los servicios desplegados en un ambiente SOA. Los servicios web, de esta manera, simplifican y estandarizan los mecanismos de descripción e invocación.

3.7.2. Origen de XML

En la década del 70, Charles Goldfarb introduce el estándar SGML[30].

SGML se estableció como el estándar internacional de representación de datos [18][19][20]

Una década más tarde, Tim Berners Lee crea la *World Wide Web* (WWW), y posteriormente funda la W3C (*World Wide Web Consortium*).

Una de las primeras iniciativas de este consorcio fue la creación de una especificación para el lenguaje de marcas de hipertexto (HTML); un lenguaje basado en SGML.

HTML provee la sintaxis que permite describir la estructura y formato de un documento de texto. Su simplicidad y tamaño compacto lo convirtieron en el formato estándar de documento para la web.

HTML permite describir como la información sera presentada por un navegador web, pero no dice nada de la naturaleza de la información presentada.

A fines de los 90, se hicieron evidentes ciertas limitaciones de HTML para determinadas situaciones.

El surgimiento de los negocios electrónicos necesitaban otro formato estándar para su funcionamiento a través de internet. Surge el Lenguaje de Marcas

29 **WS-I:** http://ws-i.org

30 **SGML**: Standard Generalized Markup Language

Extensible (XML).

Un meta-lenguaje cuyo propósito fue suplementar HTML en la presentación de los datos con la habilidad de describir la naturaleza de la información presentada.

Sin XML, la información circula por internet prácticamente sin significado o contexto. XML agrega cierta "inteligencia" a los datos que se están transmitiendo, más allá de los parámetros de presentación

3.7.2.1. Describir la información

XML es la tecnología sobre la cual se construyen los servicios web. Provee la descripción, almacenamiento y formato de transmisión para el intercambio de datos a través de los servicios web [35]

A. Estructura de los documentos XML

La estructura jerárquica de los documentos XML es una de sus principales características. Cada documento tiene un elemento raíz que engloba a todos los demás elementos, los cuales, a su vez, pueden contener subelementos.

La sintaxis XML usada en tecnologías de servicios web especifica como son representados los datos, define como son transmitidos esos datos y detalla como los servicios son publicados y descubiertos.

XML es implementado usando una serie de elementos, que, a diferencia de HTML, no están predeterminados.

Un conjunto de elementos relacionados lo denominaremos *vocabulario,* y a una instancia de un vocabulario se denomina documento XML, el cual es el bloque fundamental de una arquitectura XML.

B. Beneficios en el uso de XML

XML es universalmente aceptado: ha sido aceptado como el estándar de facto para la representación de datos e interfaces entre aplicaciones.

XML provee un modelo de datos enriquecido: XML permite modelar je rarquías, listas y tipos complejos directamente en la estructura de datos.

XML es autodescriptivo: utiliza metadata que describe los elementos de datos. Esta característica es la base para los analizadores XML que utilizan estas etiquetas y esquemas asociados para identificar elementos de datos por nombre y tipo de datos.

XML es dinámico: los documentos XML están débilmente acoplados con sus esquemas. Los programas que extraen datos directamente de la estructura XML usando expresiones estándares tales como XPath, son independientes del esquema.

XML elimina formatos fijos: el orden en el cual aparecen los elementos

de datos no necesita estar prefijado. Cada elemento puede tener una longitud variable y contener un numero variable de campos.

XML es "amigable a los humanos": ademas de los beneficios anteriores, una de las principales características que ha hecho de XML una tecnología muy popular, es ser amigable a la lectura por parte de humanos. Ademas de poder ser leído mediante el uso de diferentes herramientas, tales como navegadores, editores XML o de texto, XML también es escribible, generalmente a partir del uso de herramientas de software.

Cualquier documento XML bien formado es fácilmente analizable por cualquier analizador XML estándar. Ésto no ocurre para otras representaciones de datos, tales como EDI[31] o CSV[32], que requieren el uso de analizadores especializados, o bien desarrollados "a medida".

Los documentos XML poseen mucha información sobre el contexto de los datos que contienen o información sobre la naturaleza del contenido. Los analizadores aprovechan esa metadata para recorrer el documento y obtener fácilmente los datos buscados.

XML es extensible: una aplicación y su representación de datos externa puede ser extendida sin romper el vinculo que tiene con otras aplicaciones con las cuales comparte datos. XML puede ser modificado o extendido en algunas partes sin que ésto afecte las demás porciones del documento. Es posible agregar elementos de datos y/o atributos adicionales a una porción del documento XML sin necesidad que las aplicaciones que acceden a esa porción se vean afectadas. Aquellas aplicaciones que no se interesen o necesiten los datos ampliados del documento pueden continuar su tarea sin ver afectado su funcionamiento.

Esta característica puede ser ilustrada utilizando un ejemplo simplificado de uno de los escenarios implementados en el caso de estudio planteado.

En este ejemplo usaremos el documento XML que espera recibir como entrada al Servicio XML implementado (FIGURA 3.26)

31 **EDI**: Electronic Data Interchange

32 **CSV**: Comma-separated Values

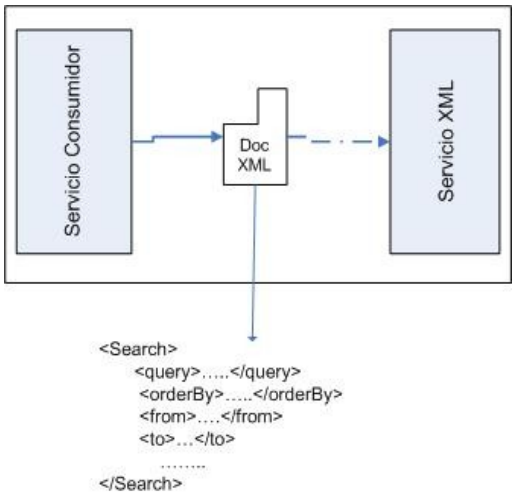


FIGURA 3.26 – Documento XML de entrada al servicio XML

Este fragmento del documento XML es extensible ya que cualquier extensión que se haga a esta porción de la estructura, no afectara la lógica del servicio XML.

En la FIGURA 3.27 se ha modificado el documento XML de entrada agregando el atributo ***maxhits*** al elemento ***query***

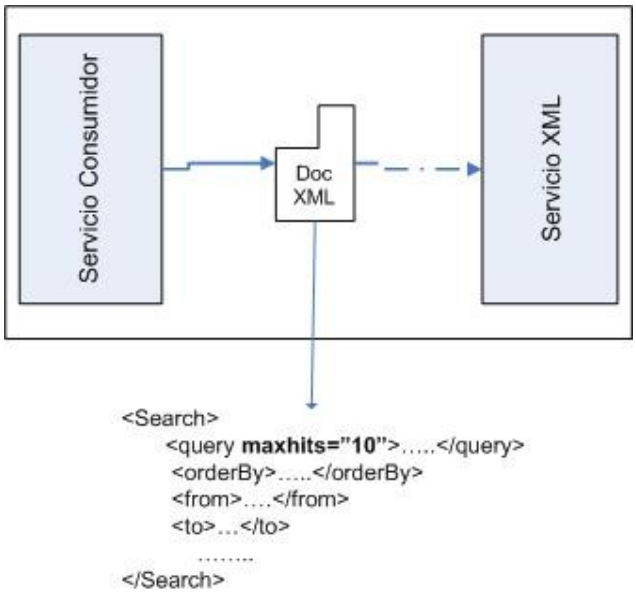


FIGURA 3.27 – Documento XML modificado

Como el **servicio XML** no contempla una lógica de procesamiento para el nuevo atributo agregado, *maxhits*, **simplemente lo ignorará y procesará el documento de la misma manera en que lo venía haciendo**. Lo importante es que el servicio no fallará ante esta ampliación a la

estructura del documento.

3.7.2.2. XML como lenguaje de integración

En SOA/ESB, los servicios son implementaciones de funciones de negocios, ya sean autocontenidas o como *wrappers* de aplicaciones heredadas.

El principal objetivo de SOA/ESB es proveer la infraestructura y condiciones necesarias para que sistemas IT diferentes sean integrados en un ambiente uniforme y estándar. La interacción y comunicación es un factor determinante en estos ambientes, y por lo tanto un punto importante en el diseño e implementación de una arquitectura orientada a servicios usando ESB

Es necesario que los datos que fluyen por el sistema SOA/ESB y que permite que los sistemas interactuen entre sí, deban estar basados en estándares abiertos, universalmente aceptados, extensibles y, principalmente, sencillos de implementar.

Estos requisitos son cumplimentados por XML, y es la tecnología utilizada por SOA/ESB para representar los datos que viajan por el bus, y permiten la comunicación de todos sus componentes.

Siendo XML la tecnología utilizada para el intercambio de información en ambientes SOA/ESB, y existiendo múltiples sistemas heterogéneos conectados al bus, surgen necesidades adicionales, que brinden funcionalidades tales como transformación a distintos formatos de datos, y ruteo dinámico basado en el contenido de los documentos XML.

A. Extensibilidad XML

Esta **característica, intrínseca** a XML, combinada con buenas practicas de acceso a los elementos de datos proveen al ESB de un modelo de datos débilmente acoplado.

Este modelo débilmente acoplado es fundamental para el diseño e implementan de un ESB, brindando beneficios para el desarrollo, despliegue y mantenimiento de la infraestructura de integración.

Esta extensibilidad posibilita que consumidores y proveedores de mensajes XML pueden ser actualizados u optimizados de manera independiente unos de otros para el manejo y acceso a la estructura XML.

También permite que aplicaciones que utilizan analizadores de documentos XML, ya sean orientados a eventos o datos, no se vean afectadas en caso que la estructura de datos sea extendida.

B. Arquitectura Genérica de Intercambio de Datos

La potencialidad de expresiones XPath y tecnologías de *parsing* de documentos XML **promueven el débil acoplamiento** entre las aplicaciones, y facilitan

mejoras en las estructura de los datos sin forzar actualizaciones simultáneas de todas las aplicaciones involucradas.

Es decir, la tecnología XML permite que se puedan realizar cambios en las aplicaciones, siempre que impliquen agregado de datos o ampliaciones a la estructura del documento, pero sin **cambiar la estructura.**

La transformación realizada mediante el lenguaje de plantillas XSL, permite reestructurar documentos XML de un formato a otro, pudiendo transformar o ampliar el contenido.

La FIGURA 3.28 muestra una visión global de un servicio de transformación basado en plantillas XSL

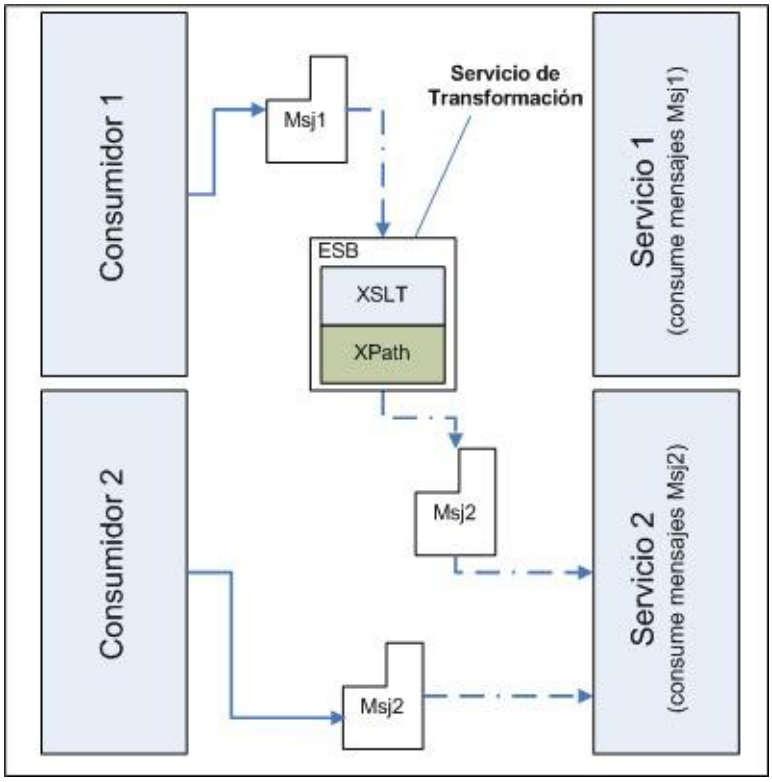


FIGURA 3.28 – Servicio de transformación basado en XSLT

El poder de transformación de XSLT, combinado con la extensibilidad de XML, permiten resolver problemas de mediana complejidad. Sin embargo, esta aproximación no funciona demasiado bien cuando el problema se complejiza.

La principal limitación es que las interacciones entre consumidores y servicios proveedores es directa, sin posibilidad que el ESB aplique reglas de negocios a los mensajes antes de llegar al destinatario original.

Es necesario un servicio de ruteo, principalmente aquellos basados en contenido.

Estos servicios de infraestructura se "incorporan" al ESB y permiten intersectar el

itinerario de los mensajes, y aplicar reglas de negocios no intrusivas, generalmente a través de expresiones Xpath.

Ampliando la FIGURA 3.28 se agrega el servicio de ruteo basado en contenido (FIGURA 3.29)

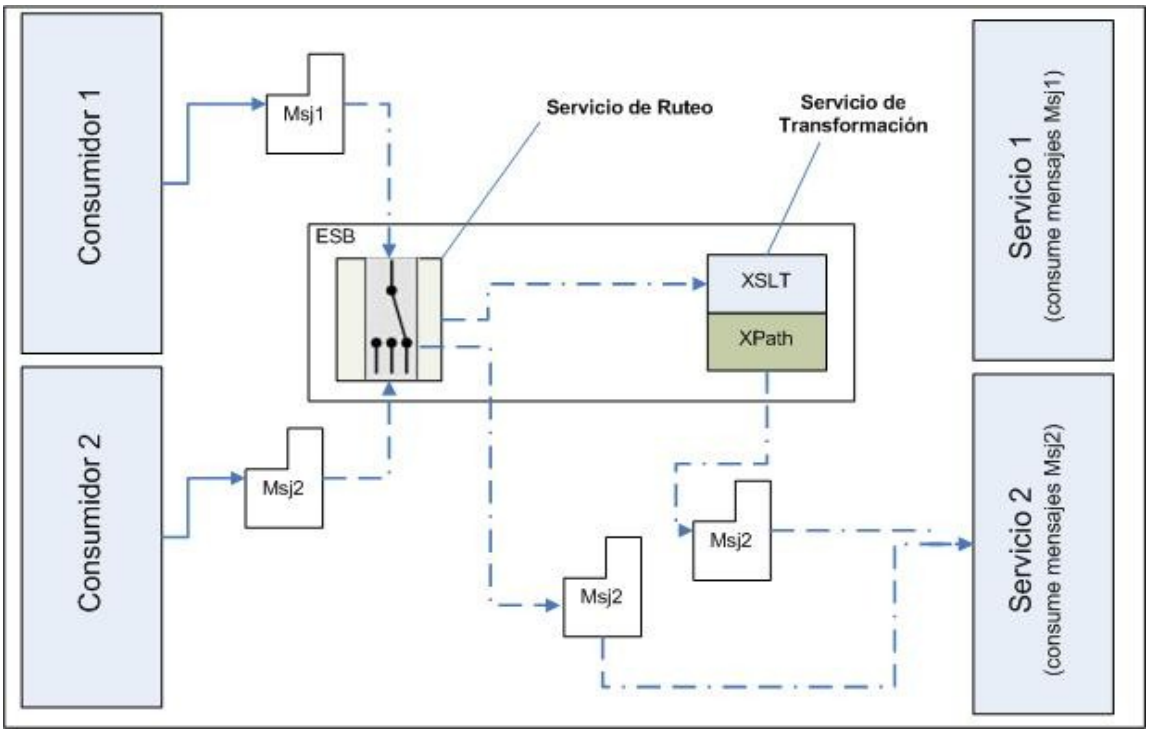


FIGURA 3.29 – Se agrega el servicio de ruteo al ESB

De esta manera, incorporando los servicios de transformación y ruteo, combinado a la potencialidad ofrecida por XML es posible cubrir un amplio rango de tareas complejas relacionadas con la integración de aplicaciones heterogéneas.

Formato canónico de datos

Una estrategia que facilita la escalabilidad de los ambientes basados en ESB es la definición y construcción de un formato de datos canónico.

Los servicios de transformación y ruteo serán esenciales para la definición y construcción del formato de datos genérico y canónico que será el tipo de datos nativo del ESB.

Todas las aplicaciones y servicios que compartan datos a través del bus usarán este formato canónico de datos. Sin embargo, ésto no significa que dichas aplicaciones/servicios deban modificar la estructura interna de sus datos, sino que el ESB será el encargado de la conversión de los datos que circulen hacia o desde el bus, a los formatos requeridos por las aplicaciones involucradas.

Esta conversión entre formatos de datos es realizada precisamente por el servicio de transformación incorporado al ESB

La FIGURA 3.30 muestra gráficamente la conversión de los mensajes intercambiados dentro del ESB a un **formato canónico**

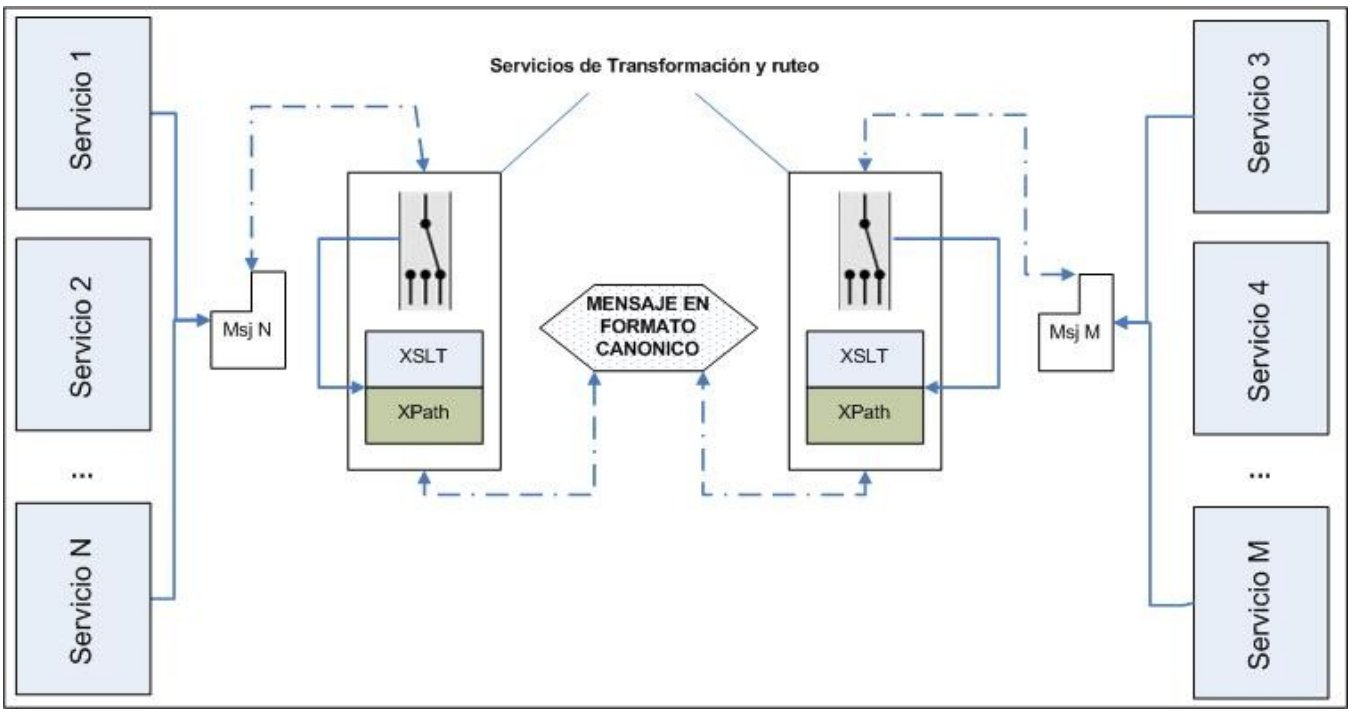


FIGURA 3.30 – Uso de un formato canónico para la comunicación en el ESB

3.7.3. Servicios Web

3.7.3.1. Orígenes

A. Mecanismo basado en XML para invocar servicios

Uno de los primeros mecanismos para "invocar" servicios remotos, utilizando tecnologías "livianas" y estándares, fue estructurar el pedido en un documento XML, enviarlo sobre algún transporte (HTTP, FTP, SMTP) a algún punto de entrada donde la aplicación del lado del servidor esté escuchando y esperando por estos documentos XML. Esta aplicación se encargaría de recepcionar el documento, interpretarlo, analizar parámetros, ejecutar el pedido y construir el documento XML de respuesta para ser enviado al cliente [21][23][24][38][39]

Esta es una de las formas más simples y primitivas de implementar SOA utilizando tecnologías livianas y estándares.

Para uno de los escenarios del caso de estudio implementado (el escenario de búsqueda basado en documentos XML) se ha desarrollado un framework para brindar soporte para la construcción de servicios XML.

B. Estándares basados en XML

El mecanismo para invocar servicios analizado anteriormente, aunque resuelve de manera sencilla el problema de integrar soluciones heterogéneas, requiere cierto **conocimiento "adicional"** para que las aplicaciones a integrar puedan interactuar entre sí. Este conocimiento hace referencia al formato del documento XML intercambiado; tanto del pedido como de la respuesta.

Esta limitación fue eliminada con el surgimiento de estándares basados en XML para la invocación/pedido, almacenamiento y descripción de servicios. Estos estándares son WSDL, SOAP y UDDI, y son la base para la construcción de servicios web.

En lo que sigue de esta sección se analizarán brevemente las principales características de los estándares WSDL y SOAP, ambos importantes en general para la construcción de arquitecturas de integración basadas en SOA/ESB, y en particular utilizados para la implementación del caso de estudio.

3.7.3.2. Describir los servicios web: WSDL

Para poder interactuar con los servicios web es necesario una descripción de los mismos que les permita a los potenciales clientes saber que datos espera recibir, si devuelve resultados, y que protocolo de comunicación o transporte utiliza.

En su forma primitiva, el acceso al servicio web se realizaba agregando información a la URL. Sin embargo esta forma tenía serias limitaciones, especialmente en la longitud máxima permitida para el envío de valores a través de la URL.

Los servicios web fueron concebidos para facilitar la implementación de soluciones distribuidas, y principalmente hacer posible la computación distribuida sobre internet.

A principios del 2001 fue presentada en la W3C[33] la especificación WSDL 1.1, creada para publicar y describir los formatos y protocolos de un servicio web de forma estándar. De esta manera, mediante el uso de interfaces estándares se evita tener que crear interacciones especiales para diferentes servidores en la web.

Los elementos WSDL contienen una descripción de los datos, generalmente mediante el uso de esquemas XML, a ser enviados al servicio web, así, tanto quien envía como quien recibe entienden los datos que están intercambiando.

Además contienen una descripción de las operaciones a ser realizadas sobre esos datos, para que el receptor sepa como procesar el mensaje, y una referencia al protocolo o transporte, así el remitente conoce como enviarlo. Generalmente WSDL es usado con SOAP, para lo cual incluye un *binding* para este protocolo

En una interacción basada en servicios web, ambas partes deben tener acceso al mismo documento WSDL así pueden entenderse mutuamente. El remitente debe conocer que formato darle al mensaje que está enviando, y el receptor debe saber

33 **W3C** (World Wide Web Consortium): http://w3.org

como interpretar correctamente el mensaje que está recibiendo. Las tecnologías de las implementaciones de los servicios web pueden ser variadas: CORBA, JMS, EJB, COM, entre otras.

Desarrolladores y usuarios utilizan WSDL para obtener una descripción formal de la interacción cliente-servicio. Durante el desarrollo, los desarrolladores utilizan los documentos WSDL como entrada de herramientas generadoras de código cliente de acuerdo a los requerimientos del servicio (FIGURA 3.31) [29][34][35]

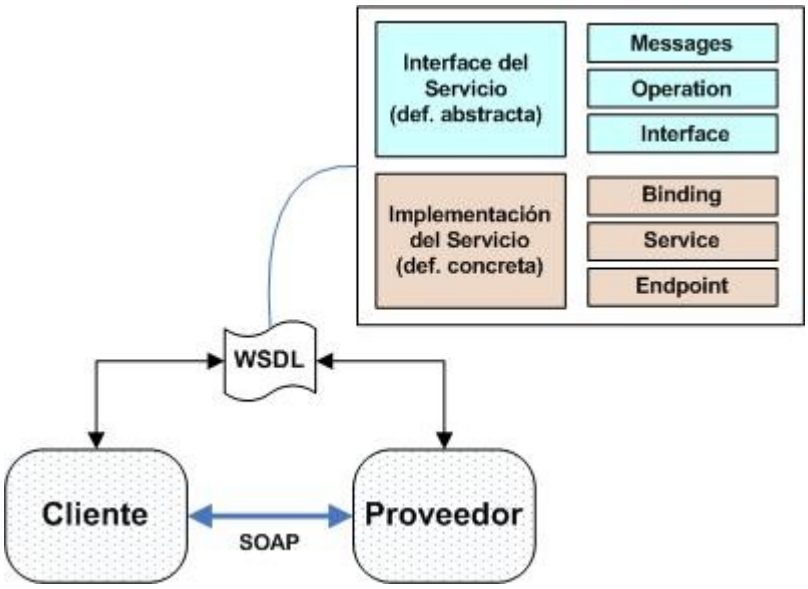


FIGURA 3.31 – WSDL compartido por clientes y proveedores

A. Elementos WSDL

WSDL define 3 elementos separados que pueden ser combinados o reutilizados:

- **Tipos**: Elemento que identifica el contenido y tipo de datos del mensaje
- **Operaciones**: Elemento que describe las operaciones realizadas sobre los mensajes
- ***Binding***: Elemento que especifica los protocolos y transportes específicos para el intercambio de mensajes por la red

Dentro de estos elementos encontramos los siguientes subelementos:

- **Types**: los tipos de datos usados en los mensajes en formato, generalmente, de esquema XML. La definición de tipos no es necesaria para tipos de datos primitivos.
- **Message**: una definición abstracta de los datos a ser transmitidos durante la comunicación.
- **PortType**: un conjunto de definiciones abstractas de las operaciones que

soporta el servicio web. Cada operación está especificada por sus mensajes de entrada y salida.

- **Binding**: protocolo y formato de datos concretos para las operaciones y mensajes, definidos para un tipo de puerto en particular
- **Port**: asocia un elemento *binding* con un solo *endpoint* de comunicación, especificado por su ubicación de red como URI (*Uniform Resource Identifier*).
- **Service**: representa un servicio web como una colección (generalmente uno) de elementos *ports*.

Las declaraciones de tipos son el elemento más abstracto en la definición de un servicio. Pueden ser definidos una sola vez y referenciados dentro de cualquier operación.

Las operaciones sobre los tipos de datos representan el siguiente nivel de abstracción, definiendo la manera en que los datos son enviados y recibidos.

El elemento *binding* representa el último nivel de abstracción y define el transporte utilizado para el envío de mensajes.

La descripción del servicio WSDL puede ser separada en 2 documentos de la siguiente manera:

La descripción de interface del servicio conteniendo la información necesaria para realizar el llamado al servicio, y está cubierta por los elementos:

- ✔ Types
- ✔ Messages
- ✔ PortType
- ✔ Binding

En otro documento la descripción de implementación del servicio informando sobre la ubicación concreta donde el servicio está publicado, a partir de los elementos:

- ✔ Port
- ✔ Service

El principal beneficio de la separación en 2 documentos de la descripción del servicio es que cada parte puede ser creada, definida, mantenida y reutilizada separada e independientemente.

B. Tipos de datos del mensaje

Los servicios web necesitan definir sus datos de entrada y salida. En WSDL los tipos son documentos XML, o parte de documentos. WSDL permite definir los

tipos de datos en archivos separados para que puedan ser reusados en diferentes servicios web.

La información sobre los tipos de datos es compartida entre quien envía y recibe los mensajes. Los receptores de los mensajes deben tener acceso a la información utilizada para codificar los datos, para entender y saber como decodificarlos.

Aunque típicamente se utilizan esquemas XML para la definición de los tipos de datos en documentos WSDL, esta porción es extensible, y permite usar otros tipo de sistema. Por ejemplo, si se conoce que el destinatario del mensaje es un objeto CORBA, es posible utilizar el sistema de tipos de CORBA en reemplazo del sistema de tipos basado en esquema XML.

C. Operaciones del mensaje

La operación es definida para que el servicio web sepa como interpretar los datos, y, de ser necesario, que datos retornar como respuesta.

Las operaciones se definen en correspondencia a patrones de mensajes conocidos.

WSDL tiene 4 tipos de operaciones soportadas por los *endpoints*:

- **One-way**: el mensaje es enviado sin un requerimiento de retornar una respuesta

  Gramática para este tipo de operación

```
<wsdl:definitions .... >
      <wsdl:portType .... >
          <wsdl:operation name="nmtoken">
               <wsdl:input name="nmtoken"? message="qname"/>
          </wsdl:operation>
      </wsdl:portType >
</wsdl:definitions>
```

- **Request/Response**: similar a una interacción estilo RPC. Se envía un mensaje y se espera la respuesta del receptor.

  Gramática para este tipo de operación

```
<wsdl:definitions .... >
      <wsdl:portType .... >
          <wsdl:operation name="nmtoken"
                          parameterOrder="nmtokens">
            <wsdl:input name="nmtoken"? message="qname"/>
```

```
            <wsdl:output name="nmtoken"? message="qname"/>
            <wsdl:fault name="nmtoken" message="qname"/>*
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

- **Solicit response**: un pedido para una respuesta sin el envío de datos de entrada.

  Gramática para este tipo de operación

```
<wsdl:definitions .... >
    <wsdl:portType .... >
        <wsdl:operation name="nmtoken" parameterOrder="nmtokens">
            <wsdl:output name="nmtoken"? message="qname"/>
            <wsdl:input name="nmtoken"? message="qname"/>
            <wsdl:fault name="nmtoken" message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

- **Notification**: este tipo de operación define múltiples receptores para un mensaje; similar a mensajes broadcast.

  Gramática para este tipo de operación

```
<wsdl:definitions .... >
    <wsdl:portType .... >
        <wsdl:operation name="nmtoken">
            <wsdl:output name="nmtoken"? message="qname"/>
        </wsdl:operation>
    </wsdl:portType >
</wsdl:definitions>
```

D. Mapeo de mensajes a protocolos

Una vez definidos los tipos de datos y los tipos de operaciones, es necesario que sean mapeados a un protocolo de transporte específico e identificar un *endpoint* o dirección donde los datos puedan ser enviados y la operación particular encontrada e invocada.

Las operaciones son agrupadas en *portTypes*, donde cada *portType* es mapeado a uno o más *ports* específicos que representan los distintos transportes sobre los cuales un servicio puede estar disponible.

D.1. PortType

Un *portType* es un agrupamiento lógico de operaciones, similar a una clase en

Java o librería en .NET.

Es posible encontrar una analogía entre las operaciones WSDL con los métodos de un objeto, siendo los mensajes WSDL análogos a los argumentos de entrada y salida.

En este caso un *portType* podría considerarse análogo a una definición de objeto que contenga potencialmente múltiples métodos.

Sin embargo, estas analogías no son totalmente exactas debido a la naturaleza extensible de WSDL cuyo propósito es proveer un nivel de abstracción mayor que el provisto por los sistemas orientados a objetos

D.2. Ports

Un *port* es usado para exponer un conjunto de operaciones (o *portType*) sobre cierto transporte. Por ejemplo, RMI es un transporte común para EJB, IIOP para CORBA, DCE (RPC) para DCOM, y otros.

En WSDL, es necesario mostrar dentro de la definición del servicio que transportes (*transport binding*) están disponibles para una colección de operaciones dada. De esta manera el sistema peticionante ubicado remotamente accederá al documento WSDL a través de una petición HTTP GET, pero puede querer interactuar con el servicio web a través de otros transportes que pueden brindarle funcionalidad adicional

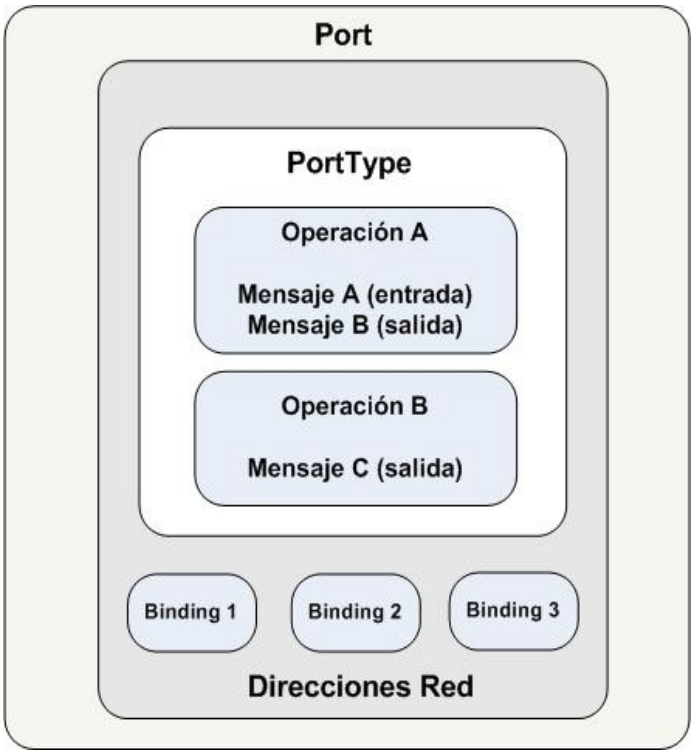


FIGURA 3.32 – Elementos de WSDL

Los *ports* por definición incluyen la/s referencia/s al transporte, que en el caso de SOAP incluye una declaración si la interacción es request/response (RPC) o pasaje de documentos (DOCUMENT).

Los *bindings* pueden definirse sobre otros transportes, como SMTP (*Simple Mail Transfer Protocol*). Esta separación de la referencia al transporte con la definición del *portType* permite que un mismo servicio web esté disponible sobre múltiples transportes sin necesidad de redefinir el documento WSDL.

D.3. Services

La parte de servicio dentro de un documento WSDL agrupa uno o varios *portTypes*.

El servicio permite elegir para un *endpoint* dado exponer múltiples categorías de operaciones para distintos tipos de interacciones. Es posible que una categoría contenga un conjunto de interacciones orientadas a documento para intercambios asincrónicos, mientras que otra contenga interacciones basadas en RPC para intercambios en tiempo real (sincrónicos)

E. Importar elementos WSDL

Es posible importar elementos WSDL ubicados en archivos distintos a través del uso de la sentencia *import*.

Los distintos elementos que conforman un documento WSDL pueden ser desarrollados independientemente y luego fusionarse para crear un documento WSDL completo que describa una instancia de un servicio web.

3.7.3.3. Acceder a los servicios web: SOAP

La mensajería XML, basada en protocolos estándares de la web, es el pilar fundamental para las arquitecturas basadas en servicios web. El estándar actual para tal mensajería XML es SOAP [22][24]

A. Introducción

SOAP originalmente fue creado en 1999 por DevelopMentor, Microsoft y UserLand Software.

Los primeros borradores fueron enviados a la W3C y fundaron las bases para la constitución del XML Protocol Working Group en Septiembre de 2000.

Actualmente el desarrollo de SOAP se encuentra en su versión 1.2, publicada por la W3C como un borrador.

SOAP es un protocolo simple y liviano basado en XML, para el intercambio de datos estructurados entre aplicaciones en ambientes distribuidos y

descentralizados, sin importar las plataformas y tecnologías subyacentes.

SOAP puede ser usado en combinación con una gran variedad de protocolos de transporte tales como HTTP, SMTP o FTP.

B. Mensajería XML usando SOAP

Los requerimientos básicos para que un nodo de red cumpla el rol de servicio web *provider* o *requestor*, en un ambiente de mensajería basado en XML y distribuido, es la habilidad para construir o analizar un mensaje SOAP, y poder comunicarse a través de una red (enviar o recibir mensajes, o ambas cosas).

SOAP es fundamentalmente un paradigma de intercambio de mensajes en un sólo sentido, sin estado; pero las aplicaciones pueden crear patrones de interacción más complejos combinando tales intercambios de un solo sentido con características proporcionadas por el protocolo utilizado y/o información específica de la aplicación en cuestión.

De esta manera es posible la implementación de patrones de interacción tales como request/response o múltiples intercambios "conversacionales" de ida y vuelta

Las interacciones SOAP ocurren entre nodos SOAP, que pueden ser remitentes o receptores de los mensajes.

Un caso especial de nodo SOAP puede ocupar el rol de intermediario entre el remitente y receptor, con el fin de manejar encabezados especiales.

C. Estructura de los mensajes SOAP

Un mensaje SOAP consta de 3 partes (FIGURA 3.33): *Envelope, Header* y *Body*.

- **Envelope** es una parte requerida y especifica el comienzo y fin de un mensaje SOAP.
- El **Header** es opcional y puede contener uno o más bloques con atributos del mensaje o definir características de calidad de servicio (QoS).

  Su intención es acarrear información de contexto especifica de las aplicaciones, tales como seguridad, identificadores de transacciones y mecanismos de correlación de mensajes.
- El **Body SOAP** es un elemento requerido dentro del elemento SOAP Envelope y contiene la información que se esta transmitiendo entre los extremos.

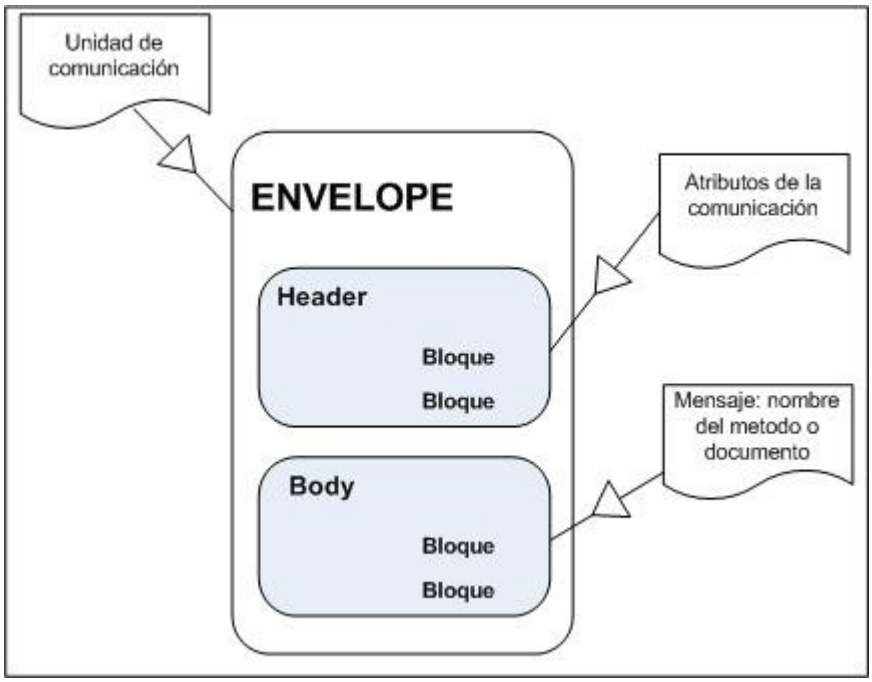


FIGURA 3.33 – Estructura de un mensaje SOAP

La elección de los datos que serán ubicados en un bloque de encabezado y los que van en el cuerpo SOAP son decisiones que se toman en el momento del diseño de la aplicación.

El punto principal que hay que tener en cuenta es que los bloques de encabezado pueden ser dirigidos a varios nodos que podrían encontrarse a lo largo del camino de un mensaje que vaya desde un remitente a un destinatario final. Tales nodos SOAP intermediarios pueden proporcionar servicios de valor añadido basados en los datos de dichos encabezados

La función del elemento Body, junto a los subelementos que posea, es la de transmitir la información entre el nodo SOAP remitente y el nodo SOAP que cumpla la función de destinatario en el camino del mensaje. La especificación SOAP no dice nada acerca del papel asumido por los nodos. El nodo SOAP destinatario final sera aquel determinado en base a la semántica especifica de la aplicación global.

D. Intercambio de mensajes SOAP

El intercambio de mensajes más simples utiliza el patrón request/response. Los primeros usos (SOAP 1.1) utilizaban este patrón como el medio para transportar llamadas a procedimientos remotos (RPC). Sin embargo, no todos los intercambios basados en el patrón request/response requieren ser modelados mediante RPC.

Existe un conjunto de escenarios que puede ser modelado como contenido basado

en XML intercambiado en mensajes SOAP para simular una conversación "ida y vuelta", en la que la semántica la proveen las aplicaciones involucradas en la recepción y envío de los mensajes.

3.7.3.4. Servicios Web y su relación con SOA

Los servicios web proveen un **mecanismo simple para integrar aplicaciones heterogéneas** (FIGURA 3.34), lo cual facilita la construcción de una infraestructura de integración basada en servicios [38][39]

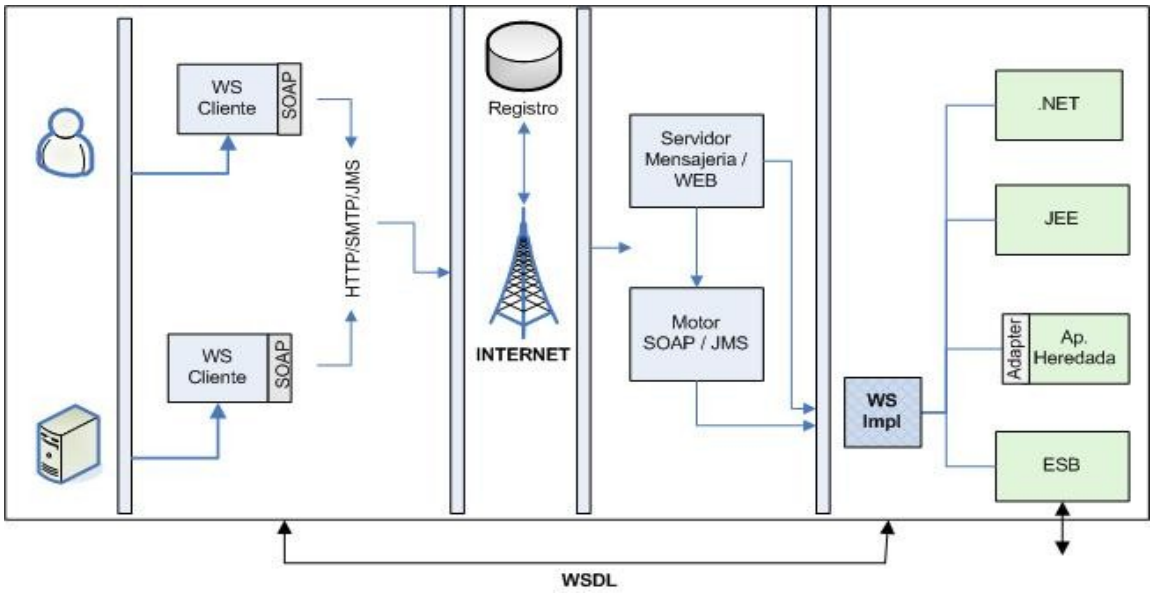


FIGURA 3.34 – Interacción cliente-proveedor mediante servicios web (adaptada de [9])

Los atributos de los servicios web destacables para la implementación de SOA se resumen en:

- **Interoperabilidad**

  Los servicios web tienen mínimos requerimientos para poder comunicar aplicaciones heterogéneas Gracias a la utilización de XML como formato de representación de información, la "invocación" básicamente significa transferir datos, sin necesidad de stubs/skeleton u otro código compilado para realizar el intercambio.

  De esta manera se logra bajo acoplamiento entre las aplicaciones involucradas.

- **Integración en tiempo real**

  Los servicios son identificados por una URL y un archivo WSDL que describe las interfaces. No se asume en ningún momento donde se está

ejecutando el servicio, por lo tanto el "binding" puede ocurrir al momento de la invocación

- **Encapsulamiento**

  Los servicios están descriptos por archivos WSDL lo cual facilita el ocultamiento de detalles de implementación a las aplicaciones que lo utilizan. Así también permite ocultar aspectos relativos a la ejecución de los servicios, tales como sistema operativo y plataforma de software/hardware donde ejecuta y lenguaje de programación en que fue programado.

  Esta característica brinda numerosos beneficios, entre los cuales figuran: flexibilidad para poder cambiar de manera transparente el ambiente de ejecución; escalabilidad al poder plantear un ambiente de clustering sin que los clientes se vean afectados; extensibilidad al poder ampliar u optimizar fácilmente la funcionalidad del servicio

- **Acceso a aplicaciones heredadas**

  Esta característica quizás sea la de mayor importancia al momento de implementar SOA. El acceso a aplicaciones heredadas puede realizarse a través de "*wrappers*" implementados con Servicios web. De esta manera se publica el WSDL para el acceso a la aplicación heredada a través del Servicio web, quien accede a la funcionalidad requerida a través de APIs nativas provistas por la aplicación

- **Basados en estándares**

  Algunos de los principales estándares analizados en este trabajo, e imple mentados posteriormente en el caso de estudio, pueden verse gráficamente en la FIGURA 3.35

FIGURA 3.35 – Principales estándares de servicios web

La implementación de estos estándares permiten ir construyendo la infraes tructura de SOA/ESB, que facilitará la integración de aplicaciones en ambientes heterogéneos

Capítulo 4

ESB (Enterprise Service Bus)

4.1. Introducción

El ESB es un concepto que puede ser analizado desde distintas perspectivas. Comúnmente se refiere a un patrón arquitectónico que describe una forma flexible e innovadora de construir una solución de integración. Desde la perspectiva de los proveedores de ESBs, éste es visto como un producto que ofrece todas las funcionalidades requeridas para la integración de aplicaciones: herramientas de desarrollo, librerías y un ambiente flexible de administración. Estos productos usualmente tienen su origen en tecnologías anteriores tales como EAI (*Enterprise Application Integration*).

Finalmente, la perspectiva que se profundiza en este trabajo, es aquella que considera al ESB como un componente fundamental en toda arquitectura orientada a servicios.

Desde la visión de SOA, el ESB es la pieza clave utilizada para la construcción de la plataforma de integración que permite que aplicaciones y fuentes de información diversas puedan exponerse como servicios e integrarse de manera natural en una infraestructura basada en estándares abiertos y tecnologías modernas, tales como mensajería o servicios web.

De esta manera, el ESB surge como la mejor opción para implementar una estrategia débilmente acoplada que facilite la integración de servicios heterogéneos, al proveer una combinación de capacidades de infraestructura, implementadas con tecnología de middleware, que permiten a una organización integrar servicios utilizando SOA, ademas de soportar un amplio rango de otras tecnologías y estrategias de integración [25]

4.2. ESB: Evolución

El surgimiento de los ESBs ha sido producto de un proceso de evolución que tiene, fundamentalmente, a EAI y servicios web como las principales tecnologías sobre las cuales se ha basado, pero que también ha recibido influencia de otras, tales como MOM, Servidores de Aplicación, XML, JMS, y en general, tecnologías relacionadas con internet [27][44][49]

El ESB, básicamente, evolucionó a partir de "tomar" las mejores funcionalidades y buenas practicas de las tecnologías que lo precedieron, y eliminando, o al menos minimizando, las limitaciones de éstas.

Sin embargo esta evolución no fue un proceso azaroso, sino el resultado de múltiples proveedores y consultores independientes trabajando conjuntamente para construir los cimientos de una tecnología de integración que se base en estándares, principalmente abiertos, y que siga los lineamientos de las arquitecturas orientadas a servicios, mensajería y XML.

Gráficamente, la FIGURA 4.1 muestra estimativamente las fechas de surgimiento o mayor expansión de las principales tecnologías de las cuales el ESB se ha influenciado

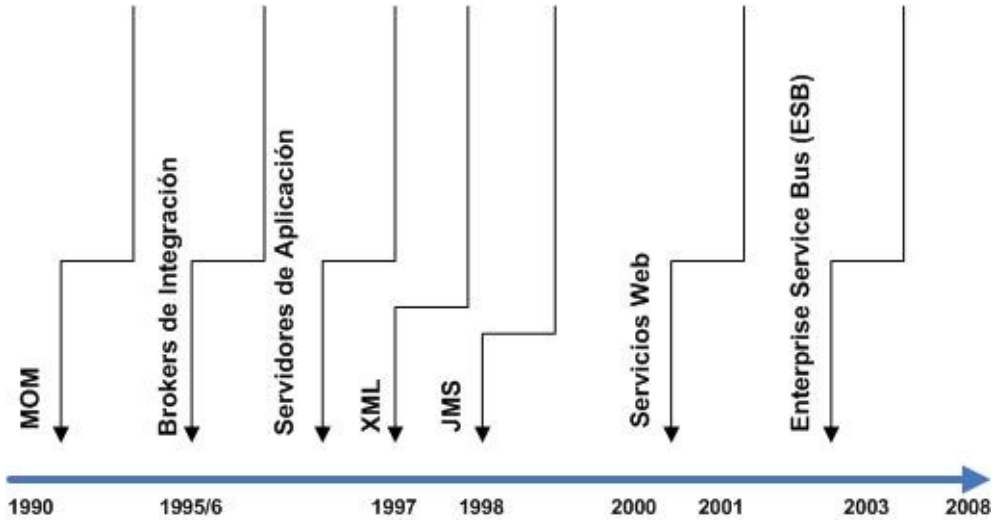


FIGURA 4.1 – Tecnologías anteriores a ESB

4.2.1. ESB y otras tecnologías

XML facilitó el surgimiento de numerosas estrategias de integración de aplicaciones. Es un mecanismo simple y portable para la representación de datos, cuya estructura puede ser descripta y validada a partir del uso de esquemas XML. Surgen los estándares SOAP y WSDL, basados en XML, que dan soporte a la

tecnología de servicios web [16][44]

El objetivo de los servicios web es conectar 2 pares, invocando servicios remotos, en interacciones punto-a-punto.

Un proveedor de servicios publica el servicio que será accedido por un consumidor de servicios, conectándose directamente al proveedor (por ejemplo, usando SOAP sobre HTTP). En este caso, el foco se centra en cuestiones relacionadas con conectividad y acceso a servicios, y no tanto en características QoS, tales como escalabilidad, confiabilidad y flexibilidad en la infraestructura de comunicación entre el proveedor del servicio y las aplicaciones clientes que lo acceden.

EAI, basada en modelos *hub&spoke* con interacciones punto-a-punto, aunque resuelva algunas cuestiones relacionadas con integración, tampoco ofrece características relacionadas con QoS, tales como las mencionadas.

Los servidores de aplicación ofrecen interoperabilidad a través de protocolos estándares, aunque no facilitan el débil acoplamiento entre los componentes que comunican, y no existe una clara separación entre la lógica de integración y de aplicación.

Por su parte, MOM permite conexiones débilmente acopladas y asincrónicas entre aplicaciones. Sin embargo al no brindar un alto grado de abstracción para definir la lógica de ruteo y transformación de datos hace necesaria cierta codificación en las aplicaciones.

En las arquitecturas IT actuales existen un amplio rango de plataformas de aplicación diferentes, posiblemente física y geográficamente distribuidas. En estos ambientes heterogéneos y distribuidos la tecnología de integración necesita de componentes de infraestructura adicionales para evitar múltiples conexiones punto-a-punto entre todas las aplicaciones involucradas (conocido como conexiones "spaghetti").

En estas arquitecturas heterogéneas y distribuidas el débil acoplamiento entre las aplicaciones es importante para facilitar una integración flexible. Es necesario que la capa de infraestructura también sea débilmente acoplada. Los servicios web, EAI e incluso las plataformas de aplicación, como JEE y .NET, fallan al proveer tal flexibilidad.

El ESB es una tecnología que ha evolucionado de los conceptos de EAI, servidores de aplicación y servicios web precisamente al eliminar algunas de sus limitaciones.

Ofrece un ambiente de integración altamente distribuido, con una clara separación de la lógica de integración y de negocios, proveyendo de la infraestructura necesaria para hospedar y acceder los servicios, y la funcionalidad que permite comunicaciones escalables, mediación y administración de servicios.

De esta manera los servicios ESBs pueden ser **configurados antes que ser codificados** [51]

La FIGURA 4.2 muestra algunas de las principales aproximaciones de integración

analizadas, y como se posiciona ESB dentro de esta clasificación.

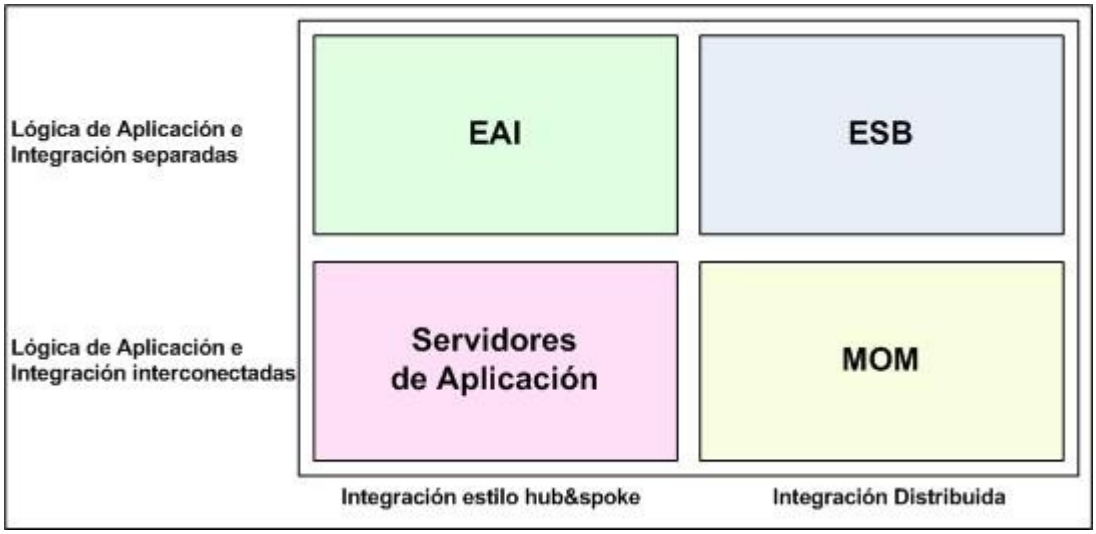


FIGURA 4.2 – Clasificación tecnologías de integración

4.3. ESB: ¿Por qué es necesario ?

Los beneficios del uso de un ESB varían de acuerdo a los problemas particulares de integración.

Sin el uso de ESB u otra tecnología similar, crear una solución de integración entre numerosas aplicaciones implicaría construir adaptadores específicos para cada aplicación que desee comunicarse e interactuar con otras.

Este modelo se conoce como arquitecturas punto a punto, las cuales son complejas y costosas en su mantenimiento. Agregar aplicaciones a la solución de integración implica crear nuevos conectores a todas las demás aplicaciones con las cuales desea interactuar.

Utilizar un ESB elimina, o al menos minimiza, los inconvenientes de las arquitecturas punto a punto, al ofrecer una plataforma de integración estándar y flexible que facilita la integración de aplicaciones en ambientes con multiplicidad de protocolos y tecnologías.

La FIGURA 4.3 muestra con un alto grado de abstracción estos dos tipos de arquitecturas

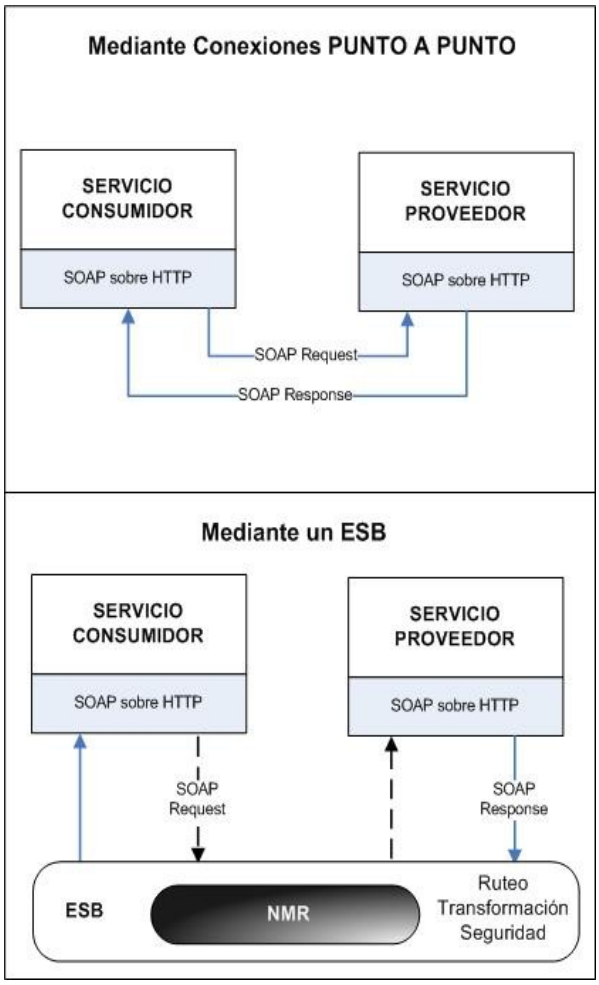


FIGURA 4.3 - Arquitectura punto a punto vs arquitectura usando ESB

El ESB provee numerosos adaptadores a distintos protocolos (JMS, SOAP, HTTP, TCP, FTP) que facilitan la incorporación de nuevas aplicaciones a la solución de integración.

Minimiza, además, el número de conexiones y por lo tanto el costo final de mantenimiento de la solución general ya que el ESB centraliza toda la lógica necesaria para la integración de las aplicaciones.

4.4. ESB: Definición

Un ESB facilita la conexión dinámica, el control y mediación de los servicios y sus interacciones. Es una infraestructura distribuida, flexible y uniforme para la administración, alojamiento y acceso a los servicios [53]

Los conceptos que definen un ESB pueden ser resumidos en:

- **Ambiente de ejecución de servicios**: los servicios necesitan ejecutarse en

ambientes que les asegure confiabilidad, seguridad y tolerancia a fallos. Este ambiente debe proveer estas características y la infraestructura para la comunicación de los servicios.

- **Servicios**: en un ESB, el único requerimiento de programación de los servicios es su implementación No es necesario código adicional para acceder o controlar los servicios, lo cual es provisto por la infraestructura del ESB. Además del soporte natural a los servicios web, los ESBs brindan conectividad con gran variedad de tecnologías, tales como componentes JEE, CORBA, MOM.
- **Bus de servicios**: framework que facilita la comunicación de los servicios de manera transparente. El bus es un componente que puede alojar y conectar aplicaciones distribuidas y servicios de infraestructura, mejorando la escalabilidad del ESB. Provee, además, la infraestructura que facilita otras características importantes del ESB, tales como enrutamiento, transformación y control del ambiente.

De esta manera, el ESB provee la infraestructura completa para la comunicación, control y administración de los servicios. Los servicios pueden estar alojados directamente en el ambiente ESB, o bien pueden ser servicios ejecutando en sistemas heredados o servidores de aplicación

Aunque no existe aún una estandarización en la definición de ESB, en general, las definiciones analizadas concuerdan en resaltar 2 de las características consideradas fundamentales en un ESB: reusabilidad e integración de servicios

Mike Gilpin [86] nos ofrece la siguiente definición de ESB:

> *"un ESB es un software de infraestructura que facilita SOA al actuar como una capa intermedia de middleware a través de la cual se hacen disponibles un conjunto de servicios reusables"*

De la definición anterior, básicamente se destacan tres conceptos importantes que constituyen los puntos sobresalientes de un ESB:

- Es una infraestructura de software
- Es un middleware
- Facilita la adopción e implementación de SOA

4.5. ESB: Características deseables

Aunque diferentes autores y bibliografía consultada brindan distintas características deseables de un ESB, en general existe un acuerdo en considerar

la lista brindada a continuación como las características fundantes de todo ESB [25][33]

A. "Omnipotente"

El ESB se ubica en el corazón de ambientes altamente distribuidos, conformados por aplicaciones y servicios heterogéneos Tienen la capacidad de adaptarse a distintos escenarios de integración.

Las aplicaciones se unen al bus cuando lo requieran, y automáticamente son capaces de interactuar y compartir datos con otras aplicaciones o servicios "escuchando" en el bus.

Aunque las interfaces de Servicios web son una parte integral de la arquitectura de un ESB, las aplicaciones no necesariamente deben ser modificadas para convertirse en Servicios web para participar en el ESB. La conectividad es lograda a través de múltiples protocolos, APIs, tecnologías de mensajería y, en general, por adaptadores "a medida" propios o provistos por terceras partes.

B. Basado en estándares

La maduración y adopción de estándares para tecnologías de integración ha ayudado al vertiginoso crecimiento y aceptación universal que han tenido los ESBs. Los ESBs hacen uso intensivo de estándares, lo cual les brinda numerosos beneficios, entre los cuales se destacan:

- Existe numerosa información y de gran calidad sobre distintos estándares utilizados por las tecnologías de integración, y en particular por los ESBs, tales como XML, WSDL, SOAP, XPath, XQuery, XSLT, XSD, por nombrar algunos. Esta abundancia de información y documentación promueven la construcción de proyectos que utilicen ESBs como tecnología de integración, y elimina las dependencias de consultores especializados en tecnologías privativas.

- Elimina o reduce el bloqueo de proveedores. El uso de tecnologías privativas para la construcción del ESB limita las posibilidades de incorporar nuevas tecnologías o adaptar las existentes ya que el *know-how* lo mantiene el proveedor. Un cambio del proveedor implicaría posiblemente la reconstrucción de la solución de integración, y recomenzar el proceso de aprendizaje.

- Los ESBs implementados con tecnología Java, ademas se benefician de las especificaciones surgidas en la JCP[34], que facilitan la interoperabilidad entre distintos componentes de software.

34 **JCP**: Java Community Process (jcp.org)

- El uso de estándares facilita la interoperabilidad entre organizaciones y proyectos que utilicen ESBs, principalmente proyectos FLOSS

Un ESB puede utilizar componentes JEE estándares tales como JMS para conexiones MOM, o JCA para conectarse con adaptadores de aplicaciones. Incluso es posible la integración con aplicaciones construidas con .NET, C/C++, C# y otras tecnologías, y en general, puede integrarse fácilmente con componentes de software que tengan soporte para SOAP y APIs de servicios web.

C. Distribuido

El ESB incorpora de los tradicionales *brokers* EAI funcionalidades que proveen servicios de integración para transformación y ruteo de datos, como así también el uso de adaptadores a aplicaciones heredadas. Sin embargo, en los tradicionales *brokers* estas funcionalidades eran usualmente centralizadas y monolíticas.

El ESB ofrece estas capacidades de integración como servicios individuales que pueden operar conjuntamente de manera altamente distribuída, y pueden escalar independientemente unos de otros.

D. Transformación de datos

En cualquier estrategia de integración la transformación de datos es un componente esencial que permite la conversión de formatos de datos de las aplicaciones involucradas en la integración, ya que éstas, usualmente, no comparten el mismo formato para describir datos similares.

Los servicios de transformación de datos se conectan al bus del ESB, siendo, de esta manera, accesibles para cualquier aplicación o servicio interactuando a través del ESB.

Tal como los ORMs[35] que resuelven la impedancia entre el mundo de objetos y el mundo relacional, el ESB, con los servicios de transformación, resuelve la impedancia existente entre formatos de datos diferentes de aplicaciones diversas.

E. Extensibilidad

El ESB ofrece las capacidades básicas que prácticamente cualquier proyecto de integración requiere. Estas capacidades pueden ser "aumentadas" con el agregado de capas tecnológicas utilizadas para usos más específicos De esta manera es sencillo agregar capas especializadas que permitan manejar software BPM (*Business Process Management*) para el uso, orquestación y coreografía de procesos de negocios.

Igualmente es posible acoplar al ESB componentes que permitan adaptar

35 **ORM**: Object-Relational Mapping

formatos de datos privativos a la representación XML canónica que utiliza el bus de servicios.

F. Orientado a Eventos

En ambientes SOA, con ESB como componente de infraestructura, las aplicaciones y servicios son considerados como *endpoints* abstractos que pueden responder a eventos asincrónicos. SOA y ESB proveen la abstracción necesaria para ocultar los detalles de cuestiones subyacentes relacionadas con conectividad.

La implementación de los servicios no necesitan saber sobre protocolos, cómo los mensajes son ruteados, ni los formatos de datos de otros servicios. Simplemente reciben un mensaje desde el ESB como evento, lo procesan y, eventualmente, retornan una respuesta al ESB quien se encargará de derivar el mensaje al lugar adecuado.

G. Seguridad y confiabilidad

El ESB ofrece seguridad en la interacción y comunicación entre las aplicaciones en base a tecnologías conocidas sobre seguridad.

La confiabilidad provista por el componente de mensajería facilita comunicaciones asincrónicas con entrega confiable de datos e integridad transaccional.

En el caso de estudio implementado, el componente de mensajería utilizado es Apache ActiveMQ[36], el cual ofrece todas las características QoS que un ESB requiere para su infraestructura de comunicación.

H. Autónomo y Federado

La infraestructura de integración del ESB ofrece un modelo de despliegue que permite la instalación, configuración y administración local de componentes de integración y adaptadores, mientras ofrece la posibilidad de integrarse con redes federadas de integración.

I. XML como tipo de datos nativo

XML es ideal para la representación de los datos que fluyen entre las aplicaciones a través del ESB. Los datos producidos y consumidos por las aplicaciones existen en gran variedad de formatos y distintas formas de empaquetamiento.

Aunque es posible que el ESB permita que circulen por el bus datos con distintos formatos, la representación de los datos usando XML brinda grandes beneficios.

---

36 **Apache ActiveMQ:** http://activemq.apache.org

Principalmente la capacidad de usar servicios especializados para combinar datos desde diferentes fuentes que permiten crear distintas vistas de los datos, enriquecer o cambiar el destino de los mismos de acuerdo a determinadas reglas de negocio.

J. Configurable

Es posible cambiar el comportamiento de los componentes de integración a partir de configuraciones y reglas, en reemplazo de la tradicional compilación del comportamiento introducido en el código fuente.

4.6. ESB: Infraestructura

El ESB provee una arquitectura que reúne los conceptos analizados en capítulos anteriores en una infraestructura de integración global que facilita las interacciones intra/internet de las aplicaciones/servicios.

En líneas generales, el ESB está compuesto por cuatro partes esenciales: Sistema de mensajería, tecnología de servicios web, ruteo inteligente basado en contenido y transformación de datos y de formato de datos.

De esta manera, es posible considerar al ESB como un elemento arquitectónico que se encarga de algunas complejidades de infraestructura de SOA (FIGURA 4.4), a saber: la representación, mediación y ejecución de los servicios, la comunicación entre los servicios clientes y proveedores y eventualmente la implementación de los servicios.

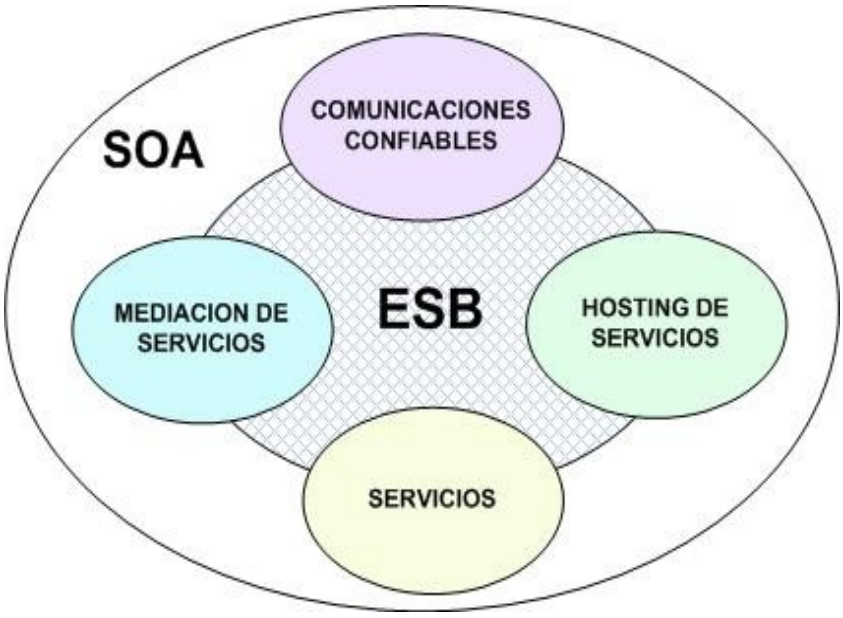


FIGURA 4.4 - ESB en el marco de una Arquitectura Orientada a Servicios (SOA)

4.6.1. ESB: Componentes de Infraestructura

El ESB posee componentes de infraestructura (FIGURA 4.5), que facilitan y permiten llevar a cabo las tareas antes mencionadas. Entre los principales componentes de infraestructura se destacan: [25][53]

A. Conectividad y Mensajería

B. Endpoints abstractos

C. Mediación y Control de Servicios

C.1. Registro de Servicios

C.2. Conversión de protocolos

C.3. Transformación de mensajes

C.4. Ruteo de mensajes

C.5. Ampliación/Modificación de mensajes

C.6 Seguridad y Confiabilidad

D. Monitoreo y Administración

E. Contenedor de Servicios

Estos componentes pueden funcionar juntos, pero no en todos los escenarios son requeridos. Diferentes escenarios pueden requerir subconjuntos específicos de dichos componentes.

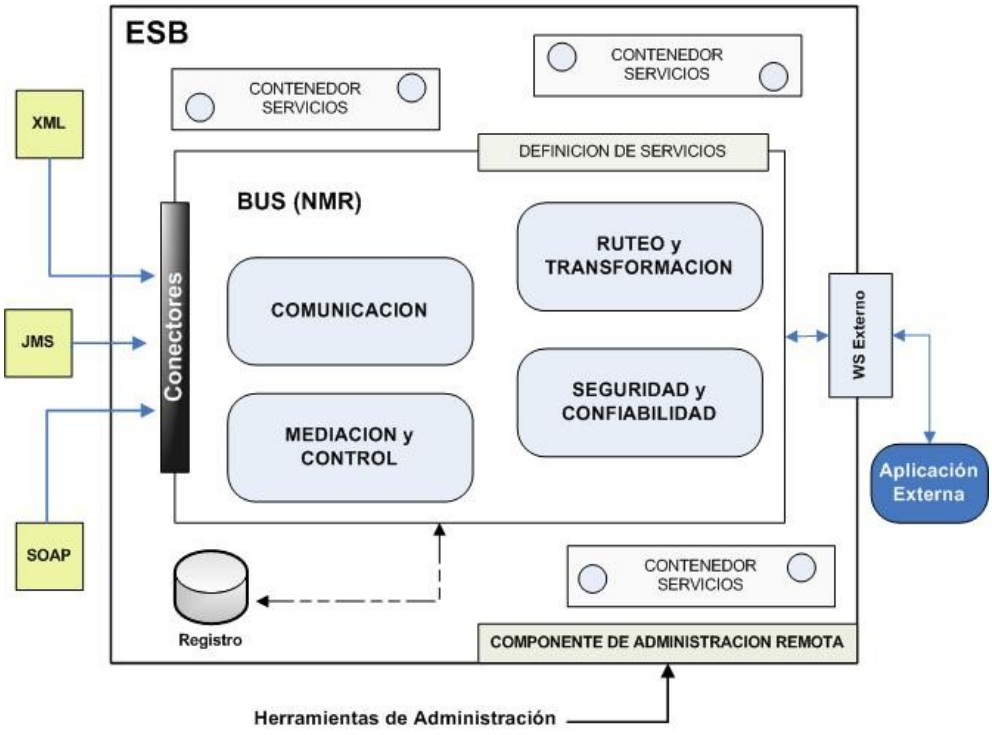


FIGURA 4.5 - Ubicación y rol de los distintos componentes de infraestructura de un ESB

A. Conectividad y Mensajería

La mensajería y conectividad son dos características fundamentales de un ESB.

La infraestructura de mensajería y conectividad del ESB no es simplemente un mecanismo para facilitar la comunicación entre dos sistemas que necesitan interactuar, tal es el caso de las soluciones de integración P2P (*peer to peer*).

Es una infraestructura que involucra otros componentes, tales como contenedores de servicios, invocaciones, transformaciones y ruteos de mensajes, de manera altamente escalable.

A.1. Mensajería

Este componente debe soportar interacciones asincrónicas, con capacidades para almacenar y posteriormente enviar los mensajes de manera confiable y segura.

El ESB puede utilizar sistemas de mensajería privativos, libres, basados en JMS, WS-Reliability, WS-ReliableMessaging o una combinación de todos, para la implementación del componente de mensajería, ya que proveen capacidades necesarias para comunicaciones asincrónicas que faciliten el desacoplamiento entre servicios consumidores y proveedores.

En ambientes donde existen múltiples aplicaciones y servicios interactuando entre si, no es práctico que cada aplicación/servicio conozca los nombres de funciones y parámetros de cada aplicación/servicio con los que necesita interactuar. Luego, las interacciones asincrónicas son un aspecto esencial para lograr interfaces débilmente acopladas, e implementar el principio homónimo requerido por SOA.

A.2 Conectividad

La FIGURA 4.6 muestra algunas de las múltiples opciones de conectividad que debe ofrecer un ESB. La diversidad en las alternativas de conectividad es una característica fundamental y distintiva de los ESBs.

Es así que aplicaciones que necesiten interactuar con la infraestructura de integración, pero que tomen sus datos de entrada de archivos planos, exporten interfaces a través de conexiones de red de bajo nivel, por ejemplo mediante *sockets,* reciban archivos XML con estructura no estándar, o bien necesiten interoperar utilizando SOAP sobre HTTP, deben poder integrarse al ESB sin requerir cambios internos significativos.

Es el ESB quien debe proveer de flexibilidad en cuanto a las tecnologías de conectividad, y no las aplicaciones involucradas en la estrategia de integración.

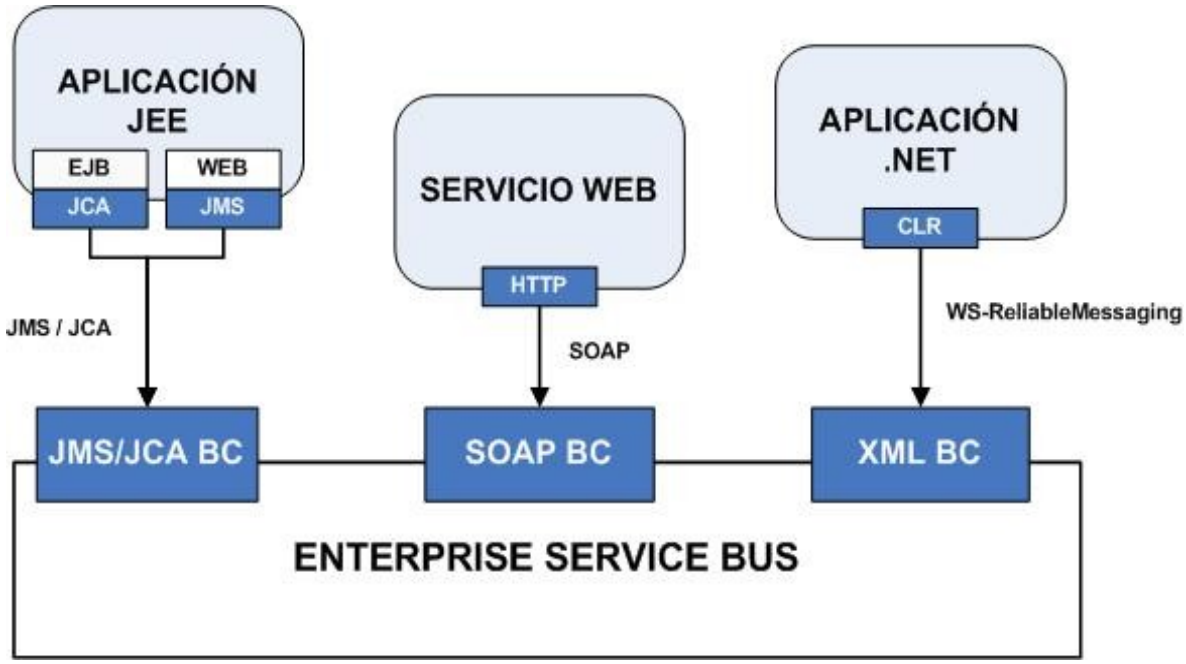


FIGURA 4.6 – Opciones de conectividad en un ESB

B. *Endpoints* Abstractos

Desde una perspectiva arquitectónica, los servicios y aplicaciones de una arquitectura de integración son tratados como *endpoints* abstractos.

Los *endpoints* se consideran abstractos ya que no es necesario conocer como están implementados, donde residen y como se conectan al ESB. Simplemente se conoce su definición de servicio, generalmente en la forma de WSDL.

Lo que un *endpoint* representa es diverso. Puede representar una operación discreta que realice un procesamiento muy especifico; representar una aplicación monolítica heredada; representar un conjunto de aplicaciones/servicios autónomos integrados mediante otras tecnologías de integración (*brokers*, servidores de aplicación) o bien, puede representar la interface a sistemas IT completamente independientes, incluso otros proyectos FLOSS, con su arquitectura de herramientas colaborativas propias.

Todos los *endpoints* abstractos participan de la arquitectura de integración de la misma manera, ya sea que representen una operación discreta muy especifica, o una interface a un sistema IT independiente.

De esta manera, los *endpoints* son considerados abstracciones lógicas de servicios que están conectado al bus.

C. Mediación y Control de Servicios

La infraestructura de mediación actúa como intermediario entre los servicios consumidores y proveedores. Sin este componente los servicios consumidores

deberían conocer las direcciones de los servicios proveedores y accederlos directamente.

El servicio consumidor delega el mecanismo de invocación al ESB, quien se encarga de los detalles de ubicar el servicio proveedor, invocarlo, o realizar el ruteo del mensaje.

De esta manera el mantenimiento de los servicios y mecanismo de búsqueda e invocaciones de servicios se simplifica.

Es posible, además, incorporar otros servicios de infraestructura a la interacción entre consumidores y proveedores.

Entre las principales funcionalidades figuran la conversión de protocolos, seguridad, transacciones, validaciones y transformaciones de formatos de datos.

Estas funcionalidades requieren de componentes que realicen el pre-procesamiento del mensaje antes de invocar el servicio destinatario.

El framework de servicios permite abstraer la ubicación especifica del servicio proveedor, al basar la relación consumidor-proveedor en contratos bien definidos (generalmente documentos WSDL) y ubicaciones abstractas (*endpoint* o URL simple).

Esta indirección en la ubicación e invocación de los servicios facilita la mediación y control por parte del ESB.

Al ser la ubicación del servicio transparente al "exportar" una única dirección lógica, la ubicación del servicio puede ser dinámica.

De esta manera el ESB puede ofrecer características de QoS, en particular escalabilidad al poder ejecutar el servicio en contenedores de servicios adicionales, o maximizar la disponibilidad de los servicios al ser posible desviar los pedidos en caso que algún contenedor deje de funcionar.

La FIGURA 4.7 muestra gráficamente la infraestructura de mediación y control de un ESB típico

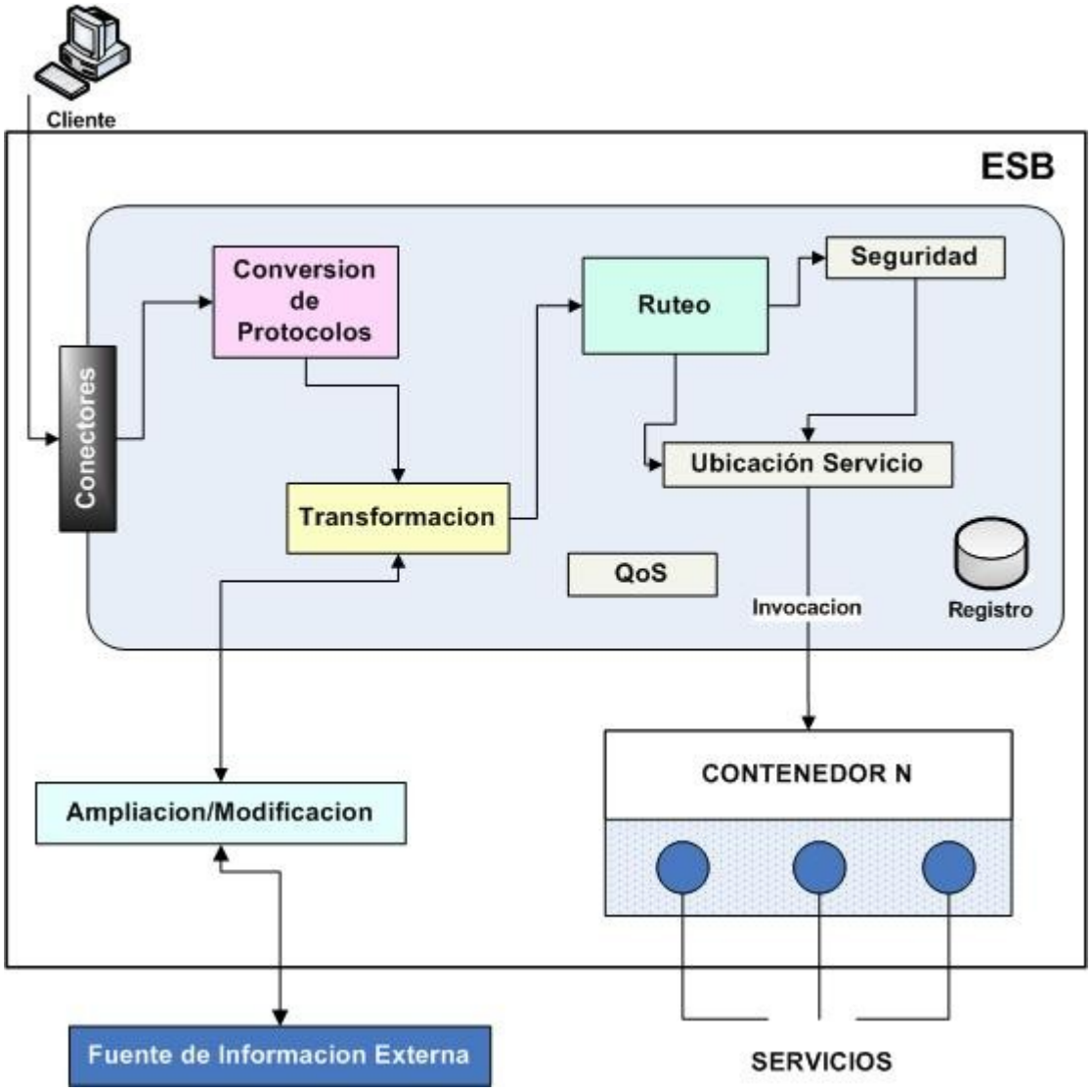


FIGURA 4.7 – Infraestructura de mediación de un ESB

Componentes de la Infraestructura de Mediación

En general estos componentes son implementados como servicios separados, lo cual permite su despliegue en cualquier parte dentro de la red, pudiendo ser actualizados, movidos, reemplazados o replicados sin verse afectadas las aplicaciones con las que coopera.

C.1. Registro de Servicios

Detrás de esta funcionalidad se esconde la idea de transparencia de ubicación El servicio consumidor se comunica con el servicio proveedor, a través del ESB, pero no conoce nada acerca de la ubicación física del proveedor. Significa ésto que el servicio consumidor está desacoplado del servicio proveedor, y que un cambio en la ubicación del proveedor no impacta la comunicación con el consumidor.

Este registro no necesariamente debe tener la complejidad de un repositorio

UDDI, pudiendo ser simplemente una base de datos o archivo de configuración XML.

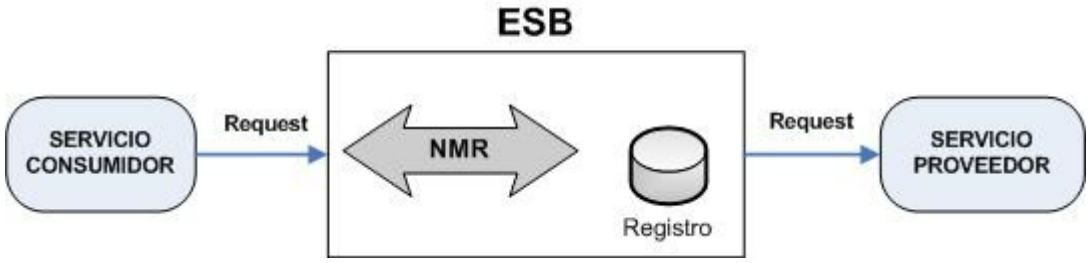


FIGURA 4.8 – Registro de servicios de un ESB

La FIGURA 4.8 muestra como el registro de servicios, dentro del ESB, abstrae al servicio consumidor la ubicación del servicio proveedor. Cambios en la ubicación afectan únicamente al registro de servicios.

C.2. Conversión del protocolo de transporte

Esta funcionalidad es útil en caso que los protocolos de transporte del servicio consumidor y servicio proveedor no coincidan. En tal caso es necesaria cierta lógica en el ESB que realice la conversión de protocolos de transporte entrantes a diferentes protocolos de transporte de salida (FIGURA 4.9)

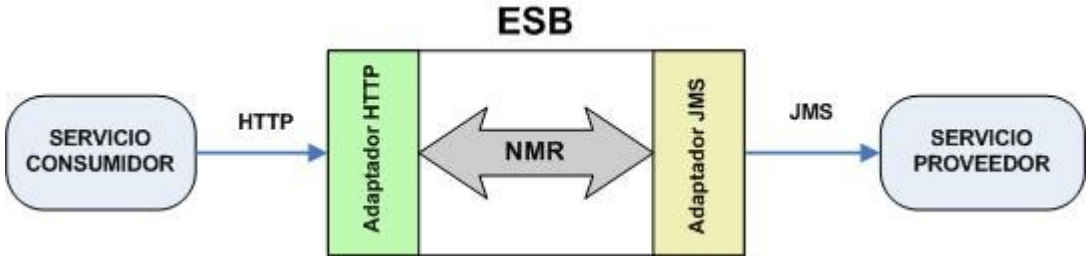


FIGURA 4.9 – Conversión de protocolo de transporte HTTP a JMS

C.3. Transformación de mensajes

La heterogeneidad en los sistemas IT, principalmente en sus formato de datos, involucrados en la estrategia de integración hacen necesarias ciertas transformaciones y adaptaciones para que puedan interactuar entre sí.

Una de tales transformaciones es en los formatos de datos en que envían y esperan recibir la información necesaria.

Esta transformación puede ser realizada por los propios sistemas que desean comunicarse. De esta manera el receptor de la información debe implementar un

adaptador específico para cada sistema que desea comunicarse con él. Aunque esta estrategia es válida y funciona en ciertos escenarios, típicamente en ambientes P2P (*peer to peer*), carece de algunas características QoS, principalmente escalabilidad.

En ambientes débilmente acoplados esta estrategia no es conveniente ya que provoca alto acoplamiento entre las aplicaciones que interactúan Surgen estrategias de transformación más transparentes y adecuadas para estos ambientes. En particular, veremos la estrategia que mejor se adapta en entornos que implementan ESBs.

Cada aplicación conoce su propio formato de datos, mientras que el componente de integración, ESB, provee de la infraestructura necesaria para poder transformar entre distintos formatos de datos.

En particular, la implementación realizada del ESB provee de un componente de transformación basado en XSLT/XPath que permite la conversión de los documentos XML que circulan en el bus a otros formatos de datos.

Una vez transformados los documentos, deben ser enviados al servicio destinatario. Como el documento está "en mano" del ESB, es posible que el documento sea ruteado a una ubicación específica, en base a su contenido.

Antes del surgimiento de estándares, como XSLT, y del uso de ESBs, los productos EAI ofrecían esta funcionalidad a través de adaptadores privativos (de código fuente cerrado y licencias de uso restrictivas).

La FIGURA 4.10 muestra un ejemplo de la necesidad de transformar el contenido del mensaje SOAP entrante, para adaptarse a los requerimientos del servicio de búsqueda destino.

La transformación de mensajes, entonces, puede ser considerada un buen ejemplo en la evolución hacia el uso de estándares abiertos en productos de integración

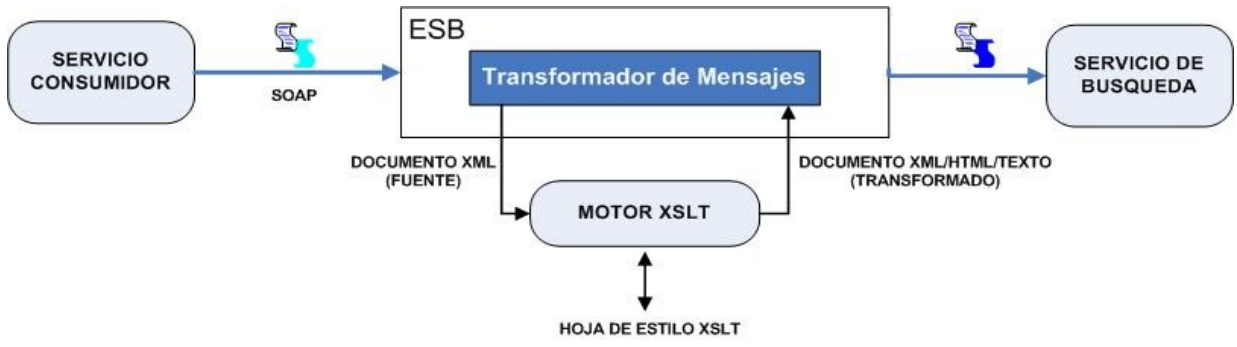


FIGURA 4.10 – Transformación de mensajes en un ESB

Combinando la extensibilidad de XML y el poder de transformación del procesamiento XSLT los servicios construidos dentro del ambiente ESB tienen

menor resistencia al cambio en situaciones complejas. Sin embargo, esta combinación no es suficiente para asegurar escalabilidad del ESB. El principal inconveniente es que requiere que los servicios tengan conocimiento directo de cada interacción punto-a-punto, y se complejiza cuando se agregan nuevos servicios.

Esta limitación puede minimizarse al introducir una nueva funcionalidad de los ESB: el **enrutamiento o ruteo** de mensajes.

C.4. Ruteo de mensajes

Rutear mensajes es una funcionalidad que permite enviar mensajes entrantes a uno o múltiples servicios proveedores. Esta funcionalidad es una clasificación de distintas formas de rutear mensajes, que incluyen, por ejemplo, ruteo basado en contenido que permite rutear el mensaje en base a su contenido, ruteo basado en filtros que evita que ciertos mensajes continúen su trayecto, o bien la capacidad de tener una lista de ruteo para el envío de mensajes a múltiples destinatarios.

La definición de un flujo de mensajes comienza encadenando varios *endpoints* de servicios, aplicando las transformaciones de mensajes requeridas, además de reglas necesarias durante el proceso de *workflow*. Estas reglas de *workflow* pueden ser definidas en términos del contenido dentro de los mensajes, o pueden estar ligadas en un *workflow* más general, conocido como enrutamiento basado en itinerario.

1. Enrutamiento basado de mensajes en **contenido**

El acto de rutear mensajes basados en el contenido generalmente es implementado a partir de componentes de *script* desplegados dentro del ESB. El script puede ser codificado en la forma de reglas BPEL[37], motores de scripting compatible con la especificación Java JSR 223 (para ESBs con soporte Java), u otros.

Existen múltiples formas de implementar el enrutamiento basado en contenido.

Por ejemplo el criterio de decisión puede ser configurado utilizando un GUI que declarativamente permita definir las reglas que actúan sobre las propiedades de los mensajes.

Puede suceder que el contenido de un solo mensaje no sea suficiente para tomar todas las decisiones necesarias para completar el itinerario desde el remitente hasta el receptor. En estos casos se utiliza enrutamiento basado en itinerario.

Gráficamente el ruteo basado en contenido puede verse en la FIGURA 4.11

37 **BPEL**: Business Process Execution Language

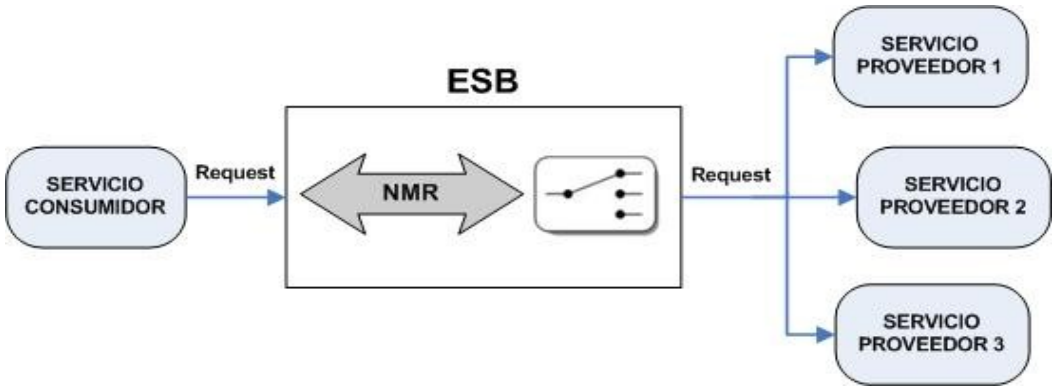


FIGURA 4.11 – Ruteo basado en contenido en un ESB

2. Enrutamiento de mensajes basado en **itinerario**

Este enrutamiento se basa en *workflows* predefinidos o bien basados en filtros.

Es similar a las cadenas de filtros en Java, en donde la respuesta de un *Servlet* se transforma en la entrada del próximo *Servlet*.

Mientras que la cadena de *Servlets* es definida mediante archivos de configuración de JEE, el enrutamiento basado en itinerario se define mediante reglas de ruteo en el ESB.

Al no estar codificadas las reglas de *workflow* en los componentes es posible modificarlas en tiempo de ejecución, sin necesidad de hacer cambios en el código

C.5. Ampliación/Modificación de los mensajes

En ocasiones no es suficiente con la transformación del formato del mensaje, sino que es necesario agregar o modificar el contenido del mensaje que será enviado a la aplicación o servicio destino.

En general ocurre cuando el contenido del mensaje incluye identificadores a elementos almacenados en algún componente externo, como puede ser una base de datos.

A partir de este identificador, el ESB puede consultar la base de datos, recuperar la información relacionada, y modificar el contenido del mensaje. De esta manera el receptor del mensaje recibe todo el contenido que necesita (FIGURA 4.12).

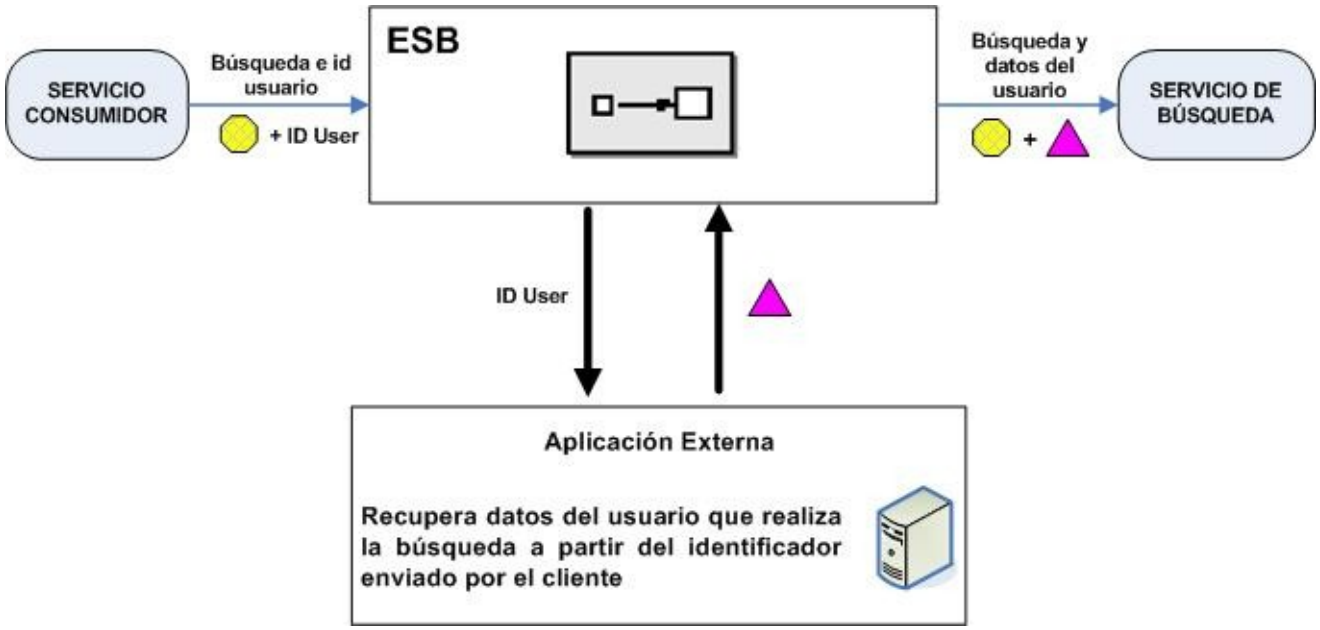


FIGURA 4.12 – Ampliación/modificación de mensajes en un ESB

C.6. Seguridad y Confiabilidad

Ya que los ESBs pueden formar parte de integración de aplicaciones de negocios críticas, debe proveer maneras de brindar estrategias de autorización y autenticación de mensajes (FIGURA 4.13). Para mensajes que pueden ser interceptados, incluso, debe proveer mecanismos de encriptado.

Generalmente el uso de ESBs se da dentro de los límites de una red interna, con lo cual se reducen los riesgos de seguridad. Sin embargo, si se ofrecen puntos de integración fuera de los límites de la red interna, la seguridad es un tema importante.

Existen múltiples tecnologías que pueden proveer seguridad al ESB.

Una simple combinación de usuario y clave podría implementarse en las colas (*queues*) definidas en la capa de mensajería del ESB. Sin embargo, los ESBs tienen otras alternativas de conectividad, como pueden ser los servicios web.

Luego, el ESB debe proveer un mecanismo de seguridad independiente del transporte.

Es posible el uso de tecnologías tales como JaaS, WS-Security u otros para proveer tal seguridad al ESB.

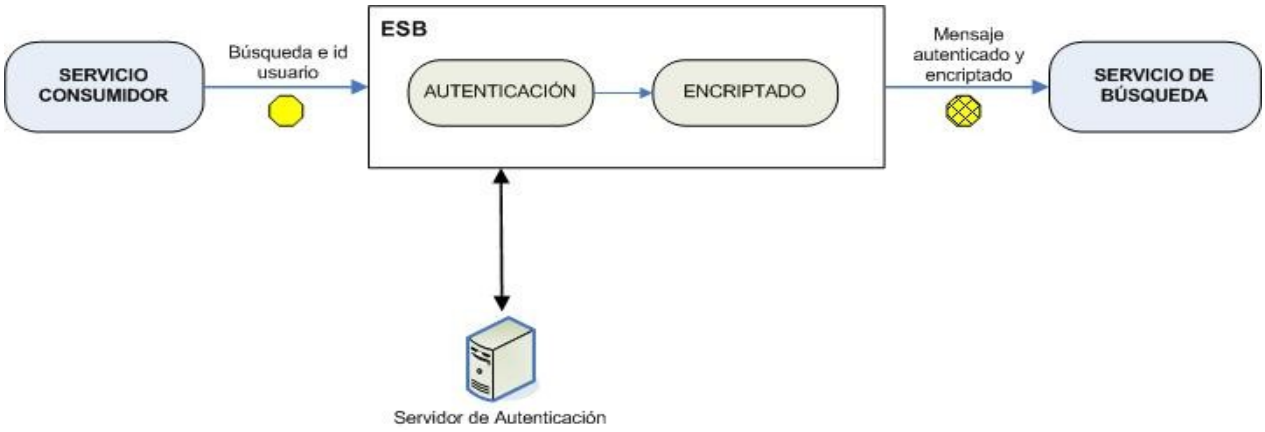


FIGURA 4.13 – Aspectos en seguridad/encriptación en un ESB

D. Monitoreo y Administración

Teniendo en cuenta que un ESB es un componente crítico en una estrategia de integración es necesario un mecanismo de administración y monitoreo.

Esta funcionalidad es similar a la provista por los servidores de aplicaciones con soporte para aplicaciones JEE.

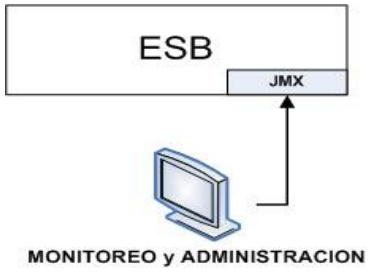


FIGURA 4.15 – Monitoreo y administración en un ESB

La funcionalidad de monitoreo y administración consiste en múltiples partes que se encargan de los distintos componentes del ESB.

Los datos de entrada de administración originados desde una consola de administración remota incluyen *start, stop* y *shutdown* de los servicios desplegados en el contenedor, mientras que los datos de salida pueden consistir en seguimientos de eventos, notificaciones sobre mensajes que han salido exitosamente de los servicios, o sobre fallas ocurridas.

Estas entradas y salidas pueden ser manejadas directamente por herramientas que implementen JMX, o bien redirigidas a otras herramientas a través de protocolos tales como SNMP[38].

38 **SNMP**: Simple Network Management Protocol

E. Contenedor de Servicios

El contenedor de servicios es considerado un proceso remoto que permite alojar componentes de software. Provee la implementación de las interfaces de los servicios y de los *endpoints* abstractos. Posee ciertas similitudes con los servidores de aplicaciones, pero con el objetivo especifico de alojar servicios de integración.

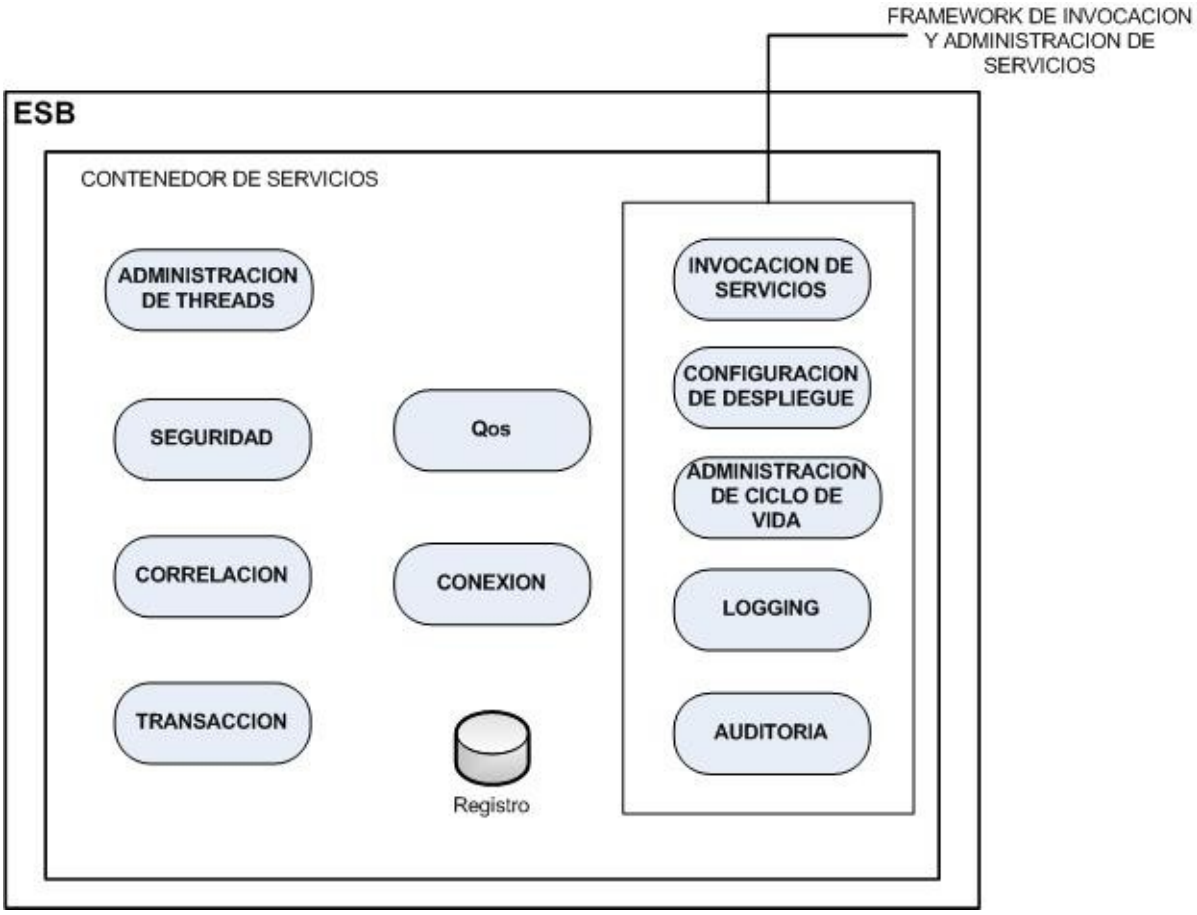


FIGURA 4.16 – Contenedor de servicios en un ESB

Sus principales características es la de ser "liviano", simple, y permitir alojar uno o más servicios en el mismo ambiente del contenedor.

Dependiendo de las capacidades de la plataforma de ESB, el mismo servicio puede estar ejecutándose sobre múltiples contenedores proveyendo de alta disponibilidad y balanceo de carga.

Capítulo 5

JBI (Java Business Integration)

5.1. Introducción

Tal como hemos analizado en capítulos anteriores, las soluciones EAI y B2B tradicionalmente requerían el uso de tecnologías no estándares para la creación soluciones de integración Esto obligaba al usuario final a permanecer "cautivo" del proveedor de tales tecnologías, o a tener que desarrollar su propia solución a medida.

Las principales desventajas de estas alternativas es que ningún proveedor puede ofrecer soluciones que abarquen el espectro de protocolos existentes, y la construcción de una solución de integración propia no ofrece posibilidades de poder integrarse fácilmente a todas las demás tecnologías

La idea impulsada por JBI es ofrecer una arquitectura basada en estándares para soluciones de integración Esta infraestructura permite el agregado de componentes de proveedores externos, y la interoperabilidad entre estos componentes de manera confiable y predecible.

Cada problema de integración es único evitando tener que permanecer cautivo de proveedores específicos, el usuario puede elegir que componentes compatibles con JBI utilizar para construir su solución de integración [43][50] [55]

5.2. JBI y tecnologías JEE

El mundo Java tiene tecnologías que promueven y facilitan la portabilidad en distintos niveles: portabilidad en el código (archivos .class), portabilidad en los

datos (JAXP y XML), portabilidad en componentes (EJB) y portabilidad en servicios (JAX-WS). Sin embargo, hasta la llegada de JBI no existía un estándar definido que permitiera la integración entre aplicaciones ejecutando sobre servidores de aplicaciones de diferentes proveedores [52][55]

El acceso, transformación e integración de datos ha sido logrado a partir de la codificación de componentes JEE, tales como EJB, JMS y JCA. Estas APIs obligaban a la codificación a bajo nivel para obtener soluciones de integración, lo cual generaba dependencias fuertes entre las aplicaciones y fuentes de datos.

Los componentes y servicios son sólo una parte de una solución global de integración. Ésta incluye además estrategias de integración, patrones de diseño adaptados para el ruteo y transformación de mensajes, auditoría de mensajes, entre otras características.

JBI se convierte en el estándar en el mundo Java para cubrir estos aspectos de las soluciones de integración

5.3. Conceptos JBI

JBI es un estándar especificado en la JSR 208 que define la manera de desarrollar aplicaciones en términos de servicios con alto grado de desacoplamiento. De manera sencilla se puede considerar que en un modelo JBI una aplicación está compuesta de servicios, y un conjunto de reglas que definen la interacción entre ellos. Los servicios son expuestos a través de los llamados componentes.

Realizando una comparación básica con la orientación a objetos, se podría pensar en los componentes similares a las clases, y los servicios a los objetos [43][54][62]

SOA y SOI[39] son los objetivos de JBI, por lo tanto JBI está construido alrededor de WSDL. Los componentes de integración pueden ser conectados al ambiente JBI usando un modelo de servicio basado en WSDL.

En arquitecturas ESB, un objetivo importante es la posibilidad de componer servicios en otros de mayor granularidad. El ambiente JBI soporta agregar múltiples definiciones de servicios en la forma de WSDL e "incorporarlos" a la infraestructura de mensajes.

Cada servicio en un ambiente JBI no se comunica directamente con los demás, sino a través de un bus de mensajes distribuido conocido como NMR (*Normalized Message Router*) y con un formato prefijado de mensaje conocido como NM (*Normalized Message*). Un mensaje normalizado consta básicamente de 3 partes:

- ✔ Un conjunto de propiedades conocidas como Message Headers
- ✔ El contenido del mensaje

39 **SOI**: Service Oriented Integration

- ✔ Un conjunto de adjuntos

Significa que una aplicación puede tener más de un servicio perteneciente al mismo componente. En una aplicación JBI todos los servicios pertenecientes a un componente en particular se agrupan en estructuras de despliegue conocidas como SUs (*Service Units*).

La especificación JBI clasifica los componentes en:

- ***Binding Components***: componentes que constituyen el punto de entrada y salida para la aplicación JBI. Llamados también adaptadores de protocolos por ser su principal función administrar la comunicación y convertir protocolos específicos a mensajes normalizados cuando funciona como punto de entrada, y mensajes normalizados a protocolos cuando funciona como punto de salida.

- ***Service Engines***: componentes que implementan la lógica de negocio, dentro de la aplicación JBI.

JSR 208 garantiza que los componentes JBI:

- Sean **portables**: los componentes son portables a través de diferentes implementaciones JBI
- Sean **administrables**: los componentes son administrados de manera centralizada
- Sean **interoperables**: los componentes deben ser capaces de proveer y consumir servicios de otros componentes, sin importar los detalles de implementación o protocolos de transporte.

Las aplicaciones JBI usualmente son conocidas como *Composite Applications* o *Service Assemblies* ya que agrupan a un conjunto de servicios relacionados entre si por una interacción o flujo que define una aplicación compleja. Estas aplicaciones son desplegadas en ESBs compatibles con JBI.

5.4. Arquitectura JBI

Como fue introducido anteriormente, JBI es un estándar Java que define un ambiente para *plugins* (en forma de componentes) que interactúan mediante un modelo basado en servicios. Principalmente estructura la integración de aplicaciones dentro de los parámetros establecidos por SOA y permite maximizar el desacoplamiento entre componentes al crear una semántica de interoperabilidad bien definida basada en estándares de mensajería

La interoperabilidad entre componentes es alcanzada a través de medios estándares de invocación de servicios basados en mensajes. WSDL es el modelo

de mensajería utilizado.

JBI define lo que comúnmente se denomina ***Contenedor de servicios*** (FIGURA 5.1) en sistemas ESB [54][62]

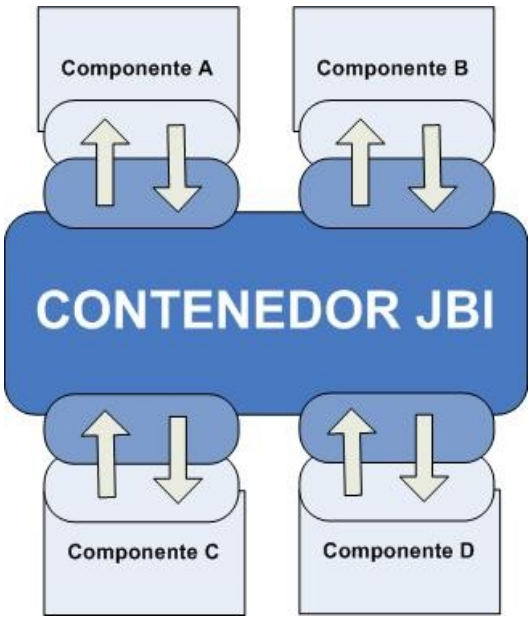


FIGURA 5.1 - Framework de *plugins* soportado por JBI (adaptada de [54])

JBI provee interfaces específicas para ser usadas por los *plugins,* mientras que éstos tienen determinadas interfaces para ser usadas por JBI.

La interacción entre componentes es asincrónica mediante el ambiente JBI. Esta separación favorece el desacoplamiento entre servicios proveedores y consumidores, lo cual, en general, es altamente deseable en SOA, y particularmente en soluciones de integración

En este modelo orientado a servicios y basado en WSDL, los componentes JBI son responsables de proveer y consumir servicios.

Al proveer un servicio el componente está haciendo disponible una o varias funciones que pueden ser consumidas por otros componentes, inclusive él mismo.

Tales funciones son modeladas como operaciones WSDL, que involucran el intercambio de uno o más mensajes.

Un conjunto de 4 MEPs[40] definen la secuencia de mensajes permitidos durante la ejecución de una operación Este comportamiento compartido, entre componentes consumidores y proveedores, sobre el patrón de intercambio de mensajes es el fundamento de la interoperabilidad entre tales componentes en JBI.

40 **MEP**: Message Exchange Pattern

La FIGURA 5.2 muestra un ejemplo de ambiente JBI dentro de una única JVM.

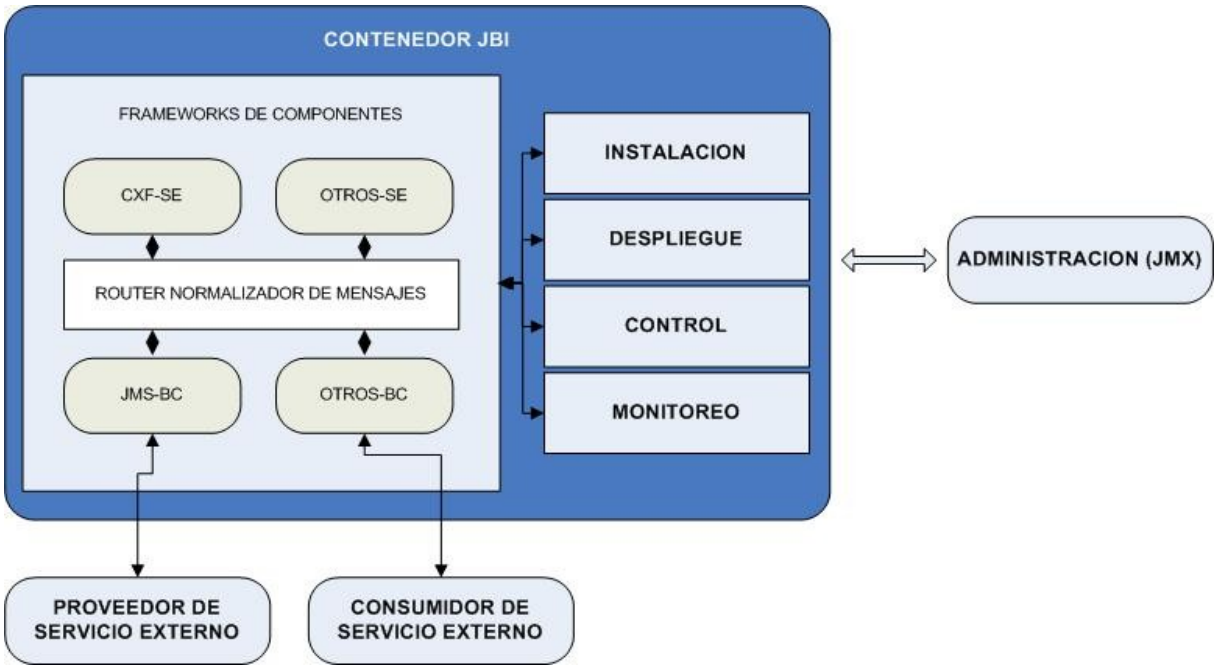


FIGURA 5.2 – Ambiente JBI (adaptada de [54])

Los servicios consumidores y proveedores se muestran gráficamente como entidades externas al ambiente JBI. Pueden usar gran variedad de tecnologías para comunicarse con los *binding components* (en la FIGURA 5.2, aquellos componentes con sufijo BC) del ambiente.

La interacción entre servicios consumidores y servicios proveedores siempre está basada en un modelo de servicio. La interface de servicio es el aspecto en común entre ellos. WSDL 1.1 o 2.0 son utilizados para definir el contrato a través de la interface de servicio.

En la FIGURA 5.2 también se pueden observar algunos de los elementos más importantes que forman parte de una arquitectura JBI: componentes SE y BC, NMR (router normalizador de mensajes), controles basados en JMX. Estos elementos se analizan en la sección siguiente.

5.5. Elementos importantes dentro de la arquitectura JBI

A. Modelo de mensajería basado en WSDL

JBI modela los servicios producidos y consumidos por los componentes utilizando WSDL 1.1 o 2.0 como lenguaje descriptor de los servicios.

WSDL provee un modelo declarativo (FIGURA 5.3) para los servicios basados en mensaje en 2 niveles:

- **Modelo de servicio abstracto**: un servicio es definido usando un modelo de mensajería abstracto, sin referencias a protocolos específicos
- **Modelo de servicio concreto**: un servicio abstracto ligado a un protocolo en particular

En JBI, el modelo de interacción entre componentes se basa en el modelo de servicio abstracto, obteniendo neutralidad en los protocolos de comunicación, pero se definen usando el modelo de servicio concreto.

JBI hace uso intensivo del modelo de mensajes abstracto. Lo utiliza para las interacciones entre componentes.

Los componentes pueden tener 2 roles en tales interacciones:

- Servicios Proveedores
- Servicios Consumidores

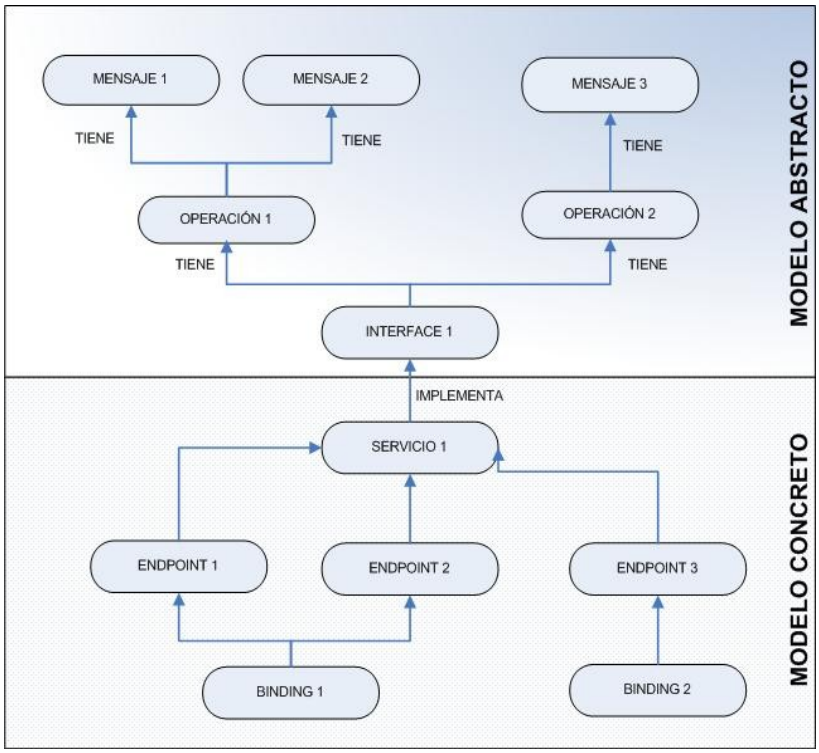


FIGURA 5.3 – Modelo WSDL (adaptada de [54])

B. Componentes JBI

Los componentes JBI son considerados contenedores: entidades donde otros componentes (*en la forma de servicios agrupados*) son desplegados a fin de configurarlos para proveer o consumir servicios particulares.

Los componentes se dividen en 2 tipos:

- **Service Engine (SE)**:

  Proveen la lógica de negocios y servicios de transformación a otros componentes.

  Esencialmente son contenedores estándares de servicios proveedores y consumidores definidos a partir de WSDL e internos al ambiente JBI.

- **Binding Component (BC)**:

  Permiten el uso de protocolos de comunicación para acceder servicios remotos, y permitir que servicios consumidores externos accedan a servicios dentro del ambiente JBI.

  Entre los protocolos más conocidos están SOAP sobre HTTP o JMS

Tanto SE como BC pueden funcionar como servicios proveedores o consumidores, o ambos. La diferencia entre ambos se basa en principios arquitectónicos, al promover la separación de la lógica de negocios de la lógica específica de comunicación, para lograr mayor flexibilidad y reducir la complejidad.

JBI soporta el despliegue directo de *configuraciones* a los componentes *plugins* usando archivos empaquetados (ZIP) llamados ***Service Units (SU)***.

El contenido de los SU no importa a la implementación JBI, excepto por un archivo descriptor, ubicado en *META-INF/jbi.xml*. Este archivo contiene información sobre los servicios estáticos que serán provistos y consumidos una vez que el SU sea desplegado satisfactoriamente en el componente. El contenido del SU es conocido únicamente por el componente en el cual es desplegado.

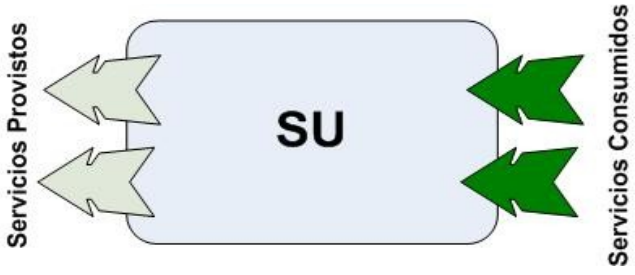


FIGURA 5.4 – Vista externa de un *Service Unit*

La FIGURA 5.4 muestra gráficamente a los SUs como "cajas negras" que declaran los servicios que estáticamente provee o consume.

Un grupo de SUs son generalmente desarrolladas o agrupadas para formar parte de una aplicación mayor, o un nuevo servicio. Este grupo de SU, junto a un descriptor de sus relaciones y componentes destinos, se denomina ***Service Assembly (SA)***.

Este SA es desplegado en el ambiente JBI. La implementación JBI se encarga de desplegar los SUs individuales en los componentes apropiados.

Un SA es un paquete (en archivos ZIP) que contiene SU individuales y un archivo descriptor llamado META-INF/jbi.xml. Este descriptor incluye información de despliegue de cada SU, y las conexiones existentes entre SUs o con servicios externos.

La FIGURA 5.5 muestra un SA empaquetado, junto a su contenido

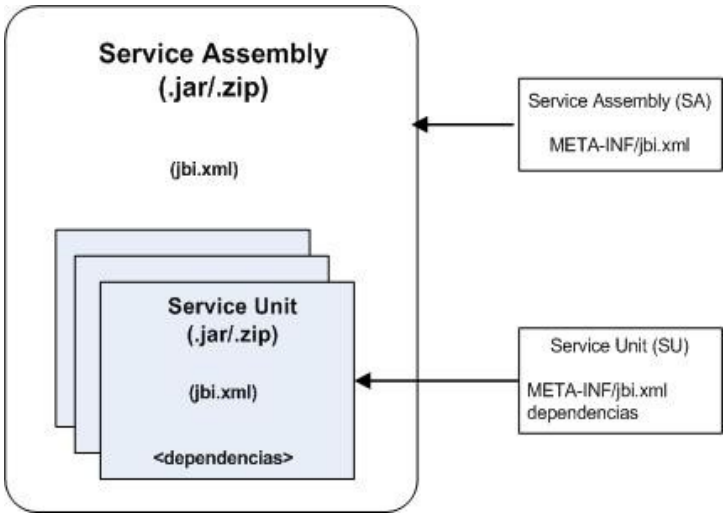


FIGURA 5.5 – Visión global de un *Service Assembly* (adaptada de [66])

C. Servicios Consumidores y Proveedores. Roles

Los componentes SE y BC de JBI funcionan como servicios proveedores, consumidores o ambos. Un servicio proveedor publica un servicio descripto por WSDL a través de un *endpoint*. Un servicio consumidor utiliza los servicios iniciando un intercambio de mensajes que invocan operaciones particulares.

Los servicios consumidores y proveedores están desacoplados, compartiendo únicamente la descripción abstracta del servicio, e interactuando a través del *Router Normalizador de Mensajes (NMR)*

Cuando interactúan realizan las siguientes responsabilidades, aunque no

necesariamente en el orden mostrado:

1. Proveedor: una vez desplegado el ambiente JBI activa el *endpoint* del proveedor.
2. Proveedor: el proveedor luego publica la descripción del servicio en formato WSDL
3. Consumidor: el consumidor descubre el servicio requerido. Ésto puede ocurrir en tiempo de diseño (binding estático) o en ejecución (binding dinámico)
4. Consumidor: invoca el servicio consultado
5. Proveedor y Consumidor: responden y envían los intercambios de mensajes en base al MEP acordado.
6. Proveedor: responde a las invocaciones a funciones
7. Proveedor y Consumidor: responden con información de estado (*fault o done*) para completar el intercambio de mensajes

Durante la activación en tiempo de ejecución, el proveedor activa los servicios que provee haciéndolos "conocidos" al NMR. De esta manera es posible rutear invocaciones de servicio a dicho proveedor.

En la FIGURA 5.6 se ve gráficamente el extracto de código que ejemplifica la activación de un *endpoint* en particular

```
public class ConsumidorComponentDemo implements ComponentLifeCycle, MessageExchangeListener {

    private ComponentContext context;
    private ObjectName extensionMBeanName;
    private MessageList messageList = new MessageList();


    public ObjectName getExtensionMBeanName() {
        return extensionMBeanName;
    }

    public void init(ComponentContext context) throws JBIException {
        this.context = context;
        //SE ACTIVA A SI MISMO
        context.activateEndpoint(new QName("http://tesis.edu.ar/", "Consumidor"), "Consumidor");
    }

    public void shutDown() throws JBIException {
    }
    public void start() throws JBIException {
    }
    public void stop() throws JBIException {
    }

    public void onMessageExchange(MessageExchange exchange) throws MessagingException {
        NormalizedMessage message = exchange.getMessage("in");
        getMessageList().addMessage(message);
    }
```

FIGURA 5.6 – Ejemplo de activación de un *endpoint* particular de un servicio consumidor

El proveedor crea un servicio descripto por un documento WSDL y lo hace disponible a través de un *endpoint.* El consumidor crea un ME (*Message Exchange*) para enviar un mensaje que invoque un servicio en particular.

D. Mensajes Normalizados

En JBI, los mensajes normalizados son los utilizados en el intercambio de mensajes y manejados por el NMR

Los mensajes normalizados constan de 3 partes

- **Mensaje XML o payload**: documento XML que adhiere a un tipo de mensaje abstracto de WSDL, sin información de protocolo o formato
- **Metadata o Encabezamiento del Mensaje (propiedades del mensaje)**: datos extras asociados al mensaje y utilizados durante su procesamiento. Puede incluir información de seguridad, transacción o personalizada.
- **Adjuntos al mensaje:** el mensaje puede contener datos adjuntos, que no necesariamente tienen que ser en formato XML. Estos datos están contenidos dentro de un manejador específico para procesarlos.

La FIGURA 5.7 muestra gráficamente un mensaje normalizado

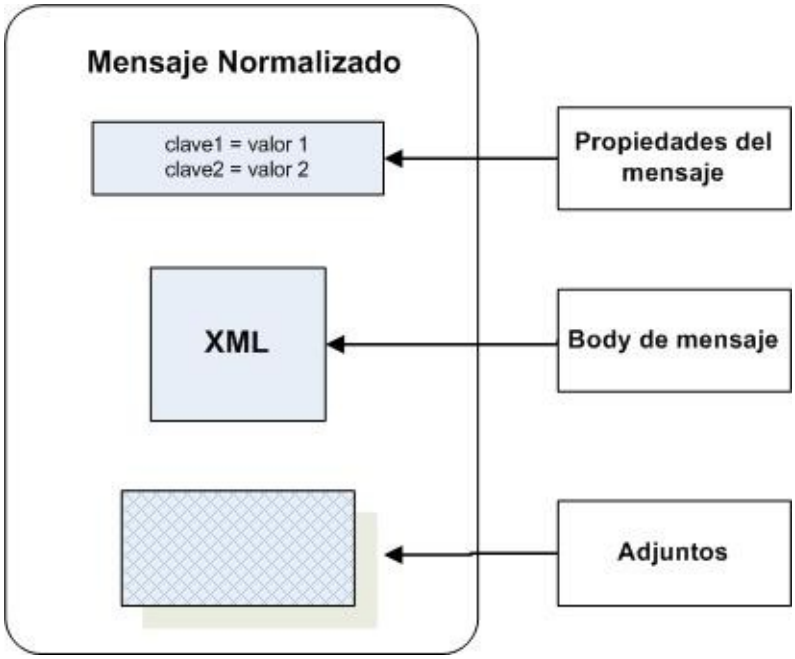


FIGURA 5.7 – Visión general de un MN (Mensaje Normalizado) (adaptada de [66])

E. NMR (Normalized Message Router)

La principal función de un ambiente JBI es rutear mensajes desde un componente a otro. Los mensajes son entregados de forma normalizada.

NMR provee la infraestructura para el intercambio de mensajes que facilita la interoperabilidad entre los componentes, desacoplando de esta manera a servicios consumidores de servicios proveedores.

A partir del NMR surgen elementos claves para el funcionamiento de un ambiente JBI

E.1. DeliveryChannel

Representa un canal de comunicación bidireccional usado por SE y BC para comunicarse con el NMR. Cada componente es provisto de un único *DeliveryChannel* para interactuar con el NMR.

E.2. Invocación de servicios y Patrones de Intercambio de Mensajes (MEP)

La invocación de servicios se refiere a la interacción entre un servicio consumidor y otro proveedor.

JBI soporta 4 tipos de interacciones

A. One-way: el servicio consumidor envía el pedido al servicio proveedor, pero no espera respuesta, ni siquiera si hubo errores.

B. Reliable one-way: el servicio consumidor envía el pedido al servicio proveedor. El proveedor puede responder con un error en caso que falle al procesar el requerimiento.

C. Request-response: el consumidor envía un pedido al proveedor y espera una respuesta. El proveedor puede responder con un error en caso que no falle al procesar el requerimiento.

D. Request optional-response: el consumidor envía un pedido al proveedor, que puede resultar en una respuesta.

Los roles de consumidor y proveedor pueden ser asumidos tanto por SE o BC, en cualquier combinación

E.3. Message Exchange (ME)

Un ME sirve como un contenedor para los mensajes normalizados que son parte de la invocación a un servicio. Encapsula los mensajes normalizados, representados por mensajes de entrada y salida del MEP utilizado, y mantiene metadata e información de estado asociada al intercambio.

Un ME es la porción local al ambiente JBI de una invocación a servicio, y está descripto por un patrón.

JBI soporta 4 patrones

1. Patrón In-Only

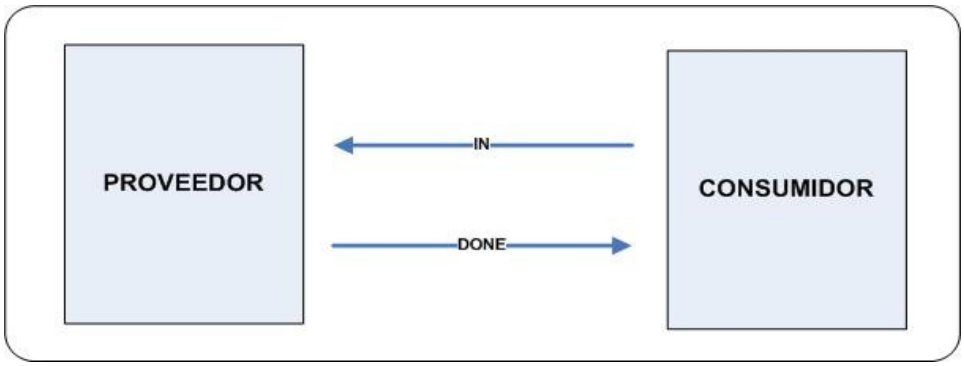


FIGURA 5.8 – Patrón In-Only

2. Patrón Robust In-Only

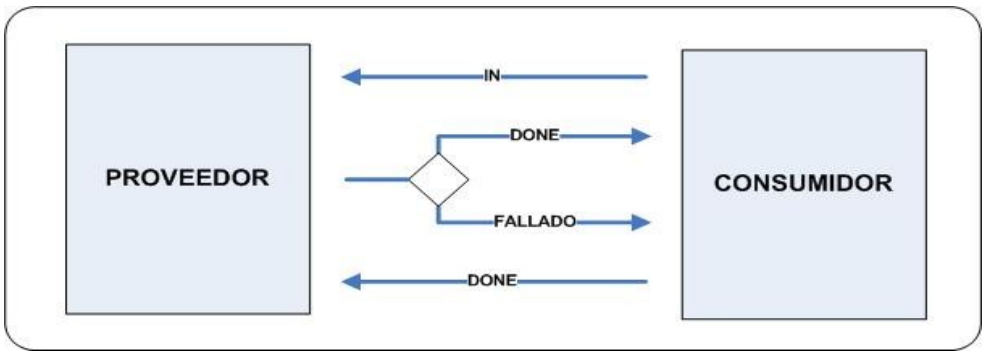


FIGURA 5.9 – Patrón Robust In-Only

3. Patrón In-Out

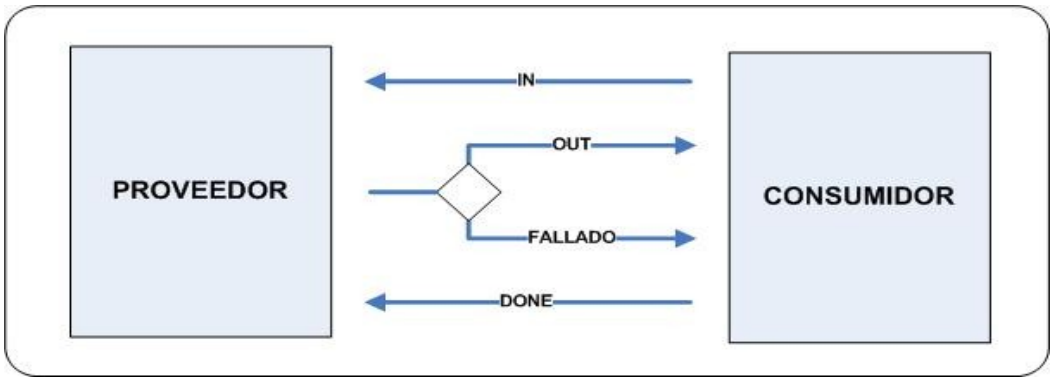


FIGURA 5.10 – Patrón In-Out

4. Patrón In optional-Out

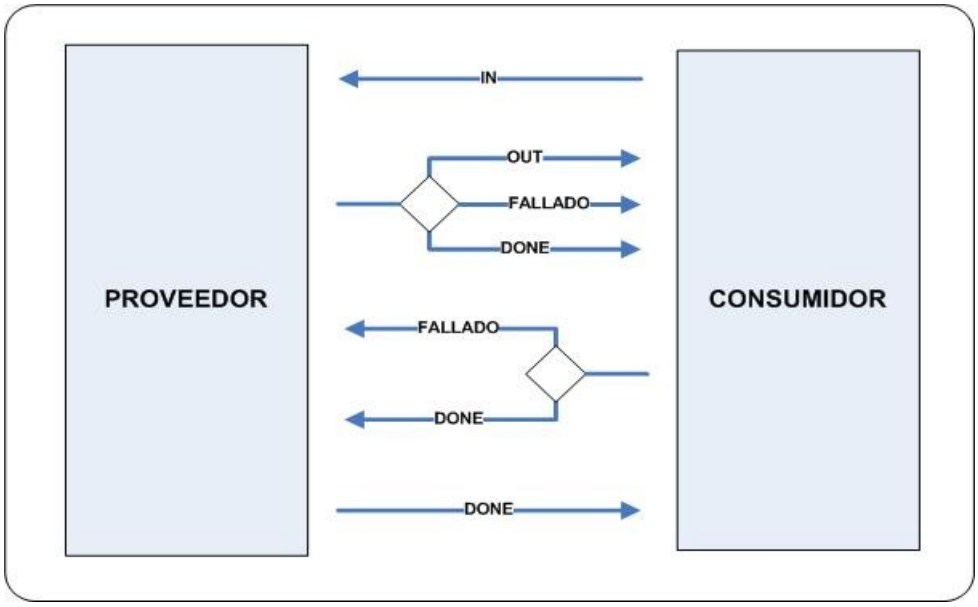


FIGURA 5.11 – Patrón In optional-Out

Cada patrón termina el intercambio con un mensaje de estado, necesario para completar el intercambio de manera determinística.

Los patrones de intercambio empiezan la interacción a partir del iniciador, que en ambientes JBI siempre es el servicio consumidor, el cual crea e inicializa una nueva instancia del intercambio (*exchange*).

En la FIGURA 5.12 puede observarse gráficamente las interacciones involucradas

en la invocación a un servicio.

En particular, en este gráfico quien inicia el intercambio es el Service Engine 1, quien asume el rol de consumidor.

El consumidor "mantiene" la instancia de ME hasta que la envía (utilizando el método *send()* del *DeliveryChannel* apropiado) a su canal para su entrega. Un componente "mantiene" un ME cuando lo acepta (*DeliveryChannel.accept()*).

Los próximos participantes alternan entre quienes aceptan el ME y lo reenvían, hasta que un participante setea el estado de intercambio a "finalizado" (*done*).

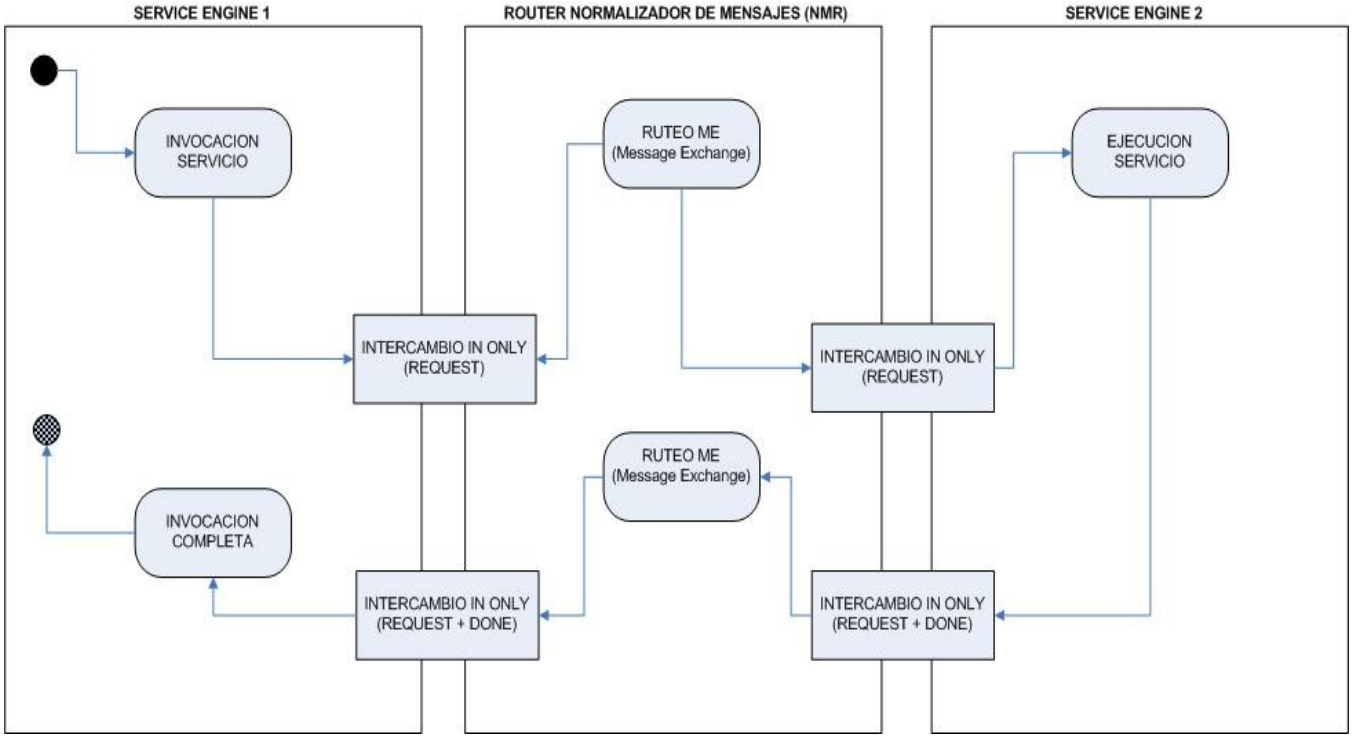


FIGURA 5.12 – Invocación a un servicio (adaptada de [54])

Algunos conceptos asociados al intercambio de mensajes:

- **Iniciador**: componente que crea el ME. Los servicios consumidores actúan como iniciadores.
- **Receptores**: componentes que atienden el ME. Los servicios proveedores actúan como receptores.
- **Roles**: los roles posibles son consumidor y proveedor.
- **Dirección**: referencia a un servicio, endpoint y nombre de operación para la dirección lógica que el NMR utiliza para rutear los mensajes.

- **Mensaje**: un ME acarrea uno o varios mensajes
- **Falla**: un ME puede acarrear como máximo una falla.
- **Estado**: describe el estado del ME: error, finalizado o activo
- **Error**: un objeto Java Exception, que indica la causa del error
- **Propiedades**: los iniciadores y receptores del ME pueden asociar propiedades a un ME. Algunos nombres de propiedades puede ser reservados por el NMR para declarar seguridad, QoS, transacción, entre otras.

Dependiendo del rol del componente en el intercambio de mensajes, la parte apropiada del mensaje es creada, inicializada y enviada al DC (*Delivery Channel*).

La FIGURA 5.13 muestra un extracto de código asociado a un intercambio In-Out

```
String addressLocalPart = null;
String addressNamespaceURI = null;
String id = null;

// ..............

javax.jbi.messaging.InOut inout =
     createInOutExchange (new QName(addressNamespaceURI, addressLocalPart), null, null);

inout.setProperty("correlationId", id);

// ..............

javax.jbi.messaging.NormalizedMessage nMsg = inout.createMessage();
inout.setInMessage(nMsg);

send(inout);
```

FIGURA 5.13 – Intercambio In-Out

E.4. *Endpoints*

Endpoints hace referencia a una dirección particular, accesible por un protocolo en particular y usado para acceder a un servicio en particular.

En JBI existen 2 tipos distintos de tipos de *endpoints*:

- **Externos**: *endpoints* fuera del ambiente JBI
- **Internos**: *endpoints* provistos por servicios proveedores dentro del ambiente JBI. Se acceden a través del API del NMR

E.5. *Bindings*

En el ambiente de integración basado en mensajería las aplicaciones que desean interactuar deben conectarse a canales de mensajería a través de *endpoints.* El proceso de conectar una aplicación o servicio a un e*ndpoint* en particular se denomina ***binding.***

Técnicamente un *binding* indica como un elemento *PortType* (interface abstracta del servicio) es ligado a un protocolo de transporte y esquema de codificación en particular.

Los BC (*Binding Component*) sirven para ligar *endpoints* internos y externos, y deben convertir los mensajes con formatos y protocolos de transporte específicos en mensajes normalizados.

Los BC y los SE (*Service Engine*) se comunican con el NMR a través de un canal (denominado *DeliveryChannel*), que permite una interacción bidireccional para el envío y recepción de mensajes.

La FIGURA 5.14 muestra gráficamente la ubicación del componente *DeliveryChannel* dentro de una arquitectura JBI, y el circuito seguido desde la llegada de un pedido

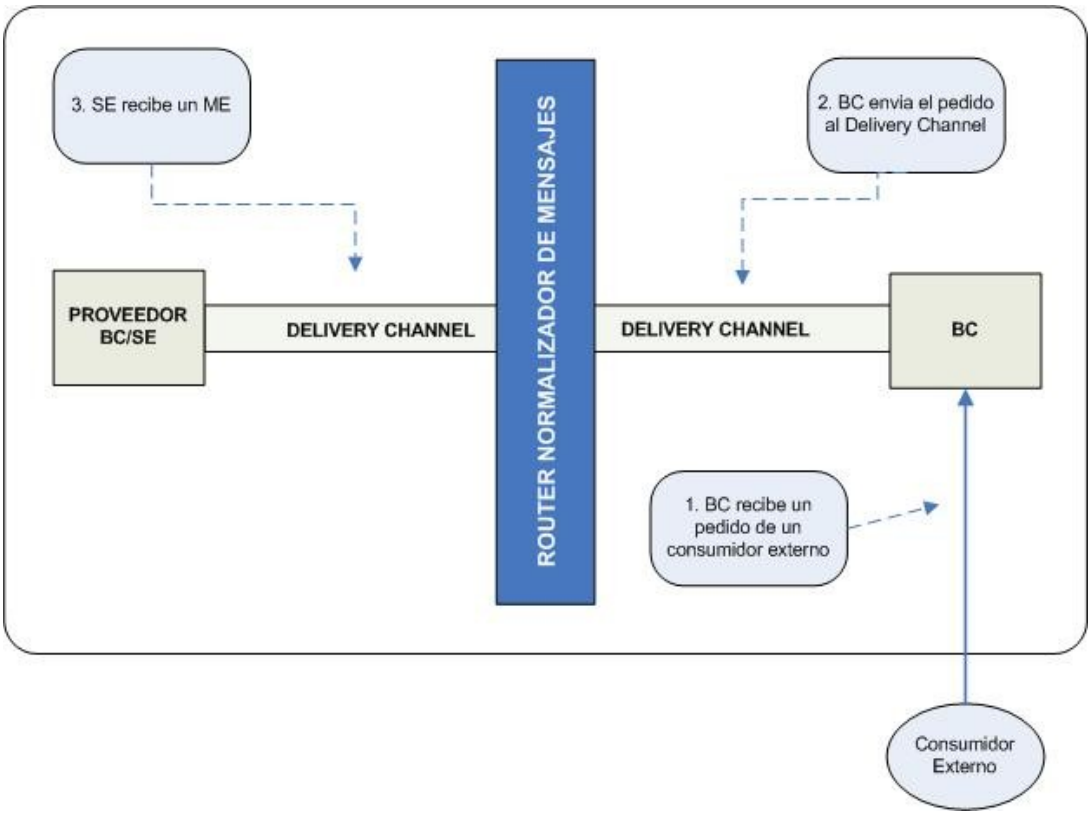


FIGURA 5.14 – Delivery Channel (adaptada de [54])

Básicamente el circuito comienza cuando el BC recibe el pedido realizado por un servicio consumidor mediante un protocolo y transporte en particular.

El BC convierte el pedido en un formato normalizado y crea un ME (Message Exchange) que funciona como un contenedor de mensajes.

Luego configura el ME (*Message Exchange*) con las propiedades que especifican el servicio y operación a invocar.

Finalmente, el BC envía al NMR el ME a través del DeliveryChannel adecuado. El NMR se encarga de rutearlo al servicio proveedor.

El NMR es clave en la creación de una solución de integración e infraestructura de servicios al desacoplar servicios proveedores de servicios consumidores, compartiendo únicamente la descripción del servicio basada en WSDL.

5.6. Administración y Monitoreo

Además del sistema de mensajería diseñado para obtener interoperabilidad entre componentes, JBI define una estructura de administración y monitoreo basada en JMX[41].

Los componentes JBI deben proveer una implementación de interfaces específicas que permitan la administración del ambiente JBI.

Las principales tareas administrativas soportadas por esta especificación son:

- Instalación de componentes JBI
- Despliegue de *assemblies* a componentes
- Iniciar y detener componentes
- Iniciar y detener grupos de servicios (*service assemblies*)

5.7. Consideraciones finales sobre JBI

A partir del análisis de ESB y JBI es posible concluir que estas dos tecnologías no son en sí mismas el fin sino el **camino** hacia SOA.

El ESB no es un componente requerido para la construcción de un SOA, ni JBI es necesario para construir un ESB o SOA. Sin embargo, el usar JBI (alguna implementación de JBI) permite construir componentes estándares que pueden

41 **JMX**: Java Management eXtensions

ser desplegados en arquitecturas ESB.

Por lo tanto, JBI **es una de las maneras** de lograr la construcción de una arquitectura orientada a servicios, y en este trabajo, por los requerimientos planteados, es la tecnología que mejor se adecua a tales pre-requisitos.

En los capítulos siguientes se plantea un caso de estudio sencillo que utiliza *jESBihca*, cuya implementación permite ilustrar desde una perspectiva netamente práctica los principales conceptos analizados hasta el momento.

Dejada intencionalmente en blanco

Parte II

jESBihca: Caso de Estudio

Capítulo 6

Comunidad FLOSS: Apache ServiceMix

6.1. Introducción

Como movimiento social FLOSS enfatiza en el concepto de **libertad** y el **compartir conocimiento**. Desde una perspectiva práctica, el desarrollo FLOSS ha producido multitud de software de altísima calidad y adopción universal, tales como el sistema operativo GNU/Linux, el servidor web Apache, base de datos MySQL o PostGreSQL. Lentamente la administración pública municipal, provincial y nacional están migrando sistemas de software basados en productos privativos a desarrollos FLOSS, y lo mismo sucede con pequeñas, medianas y grandes empresas que migran su modelo de negocio a FLOSS.

SourceForge[42] almacena en la actualidad más de 150.000 proyectos FLOSS, con una cantidad de usuarios registrados que supera 1.5 millones [a Julio 07]

En lo que sigue de este capítulo se hará un breve análisis de una comunidad de usuarios y desarrolladores de un proyecto FLOSS existente. En particular fue elegido el proyecto Apache ServiceMix por ser un proyecto joven, con una comunidad de usuarios en constante crecimiento, de la cual formo parte, y una de las principales herramientas utilizadas para la implementación del ESB requerido por el caso de estudio planteado en el capítulo siguiente.

De este breve análisis se pretende introducir como se forman las comunidades de usuarios y desarrolladores alrededor de proyectos FLOSS y se benefician de las posibilidades que brinda internet para el uso de herramientas colaborativas asincrónicas y distribuídas geográficamente, que permiten compartir conocimiento, colaborar, aprender e interactuar entre los participantes.

42 **SourceForge:** http://sourceforge.net

Luego, el capítulo siguiente analizará, **en la forma de un caso de estudio sencillo**, la problemática existente al requerir integrar búsquedas de información en las distintas herramientas colaborativas asincrónicas, específicamente aquellas usadas por muchos proyectos FLOSS.

6.2. Comunidad de Apache ServiceMix

> *"La fundación de Software Apache brinda soporte a la comunidad de proyectos de software Apache Open source. Los proyectos Apache están caracterizados por un proceso de desarrollo basado en consenso y colaboración, licencias de software abiertas y pragmáticas y un deseo de crear software de alta calidad que sea líder en su ámbito. Nos consideramos no solamente un grupo de proyectos compartiendo un servidor, sino una comunidad de desarrolladores y usuarios"*

Este párrafo extraído del sitio web de la Fundación Apache, es un excelente resumen del espíritu detrás de una comunidad de desarrolladores y usuarios de proyectos FLOSS.

Apache ServiceMix es un proyecto incorporado recientemente como proyecto de Apache[43].

Alrededor de este proyecto, como en la mayoría de los proyectos FLOSS, se está formando una enorme comunidad de usuarios y desarrolladores, geográficamente distribuídos, y que interactúan entre si a través de herramientas de software colaborativas.

En el "corazón" de la comunidad se encuentran aquellas personas, desarrolladores y analistas que concibieron la idea e iniciaron el proyecto. La maduración del proyecto favoreció que se sumen más desarrolladores y usuarios activos que permanentemente modifican el software, lo usan como plataforma de desarrollo, y reportan los errores que van surgiendo. Existen entre los usuarios activos aquellas personas que instalan y usan el software, a veces sin modificar, e incluso sin leer, el código fuente.

La comunidad de Apache ServiceMix colabora y se nutre constantemente del conocimiento generado por otras comunidades formadas alrededor de proyectos FLOSS utilizados por Apache ServiceMix. En particular las comunidades de Apache ActiveMQ, Apache Camel, Apache CXF, Spring Framework, Apache XBean, Geronimo, y en general, de las comunidades y grupos de usuarios del lenguaje Java.

---

43 **Proyecto de Apache:** http://projects.apache.org/indexes/pmc.html#Apache ServiceMix

Una hipótesis sobre el crecimiento de los proyectos FLOSS en general, y en particular el nivel de calidad del código producido por Apache ServiceMix, se debe a la capacidad adquirida de explicitar, organizar y estructurar el conocimiento adquirido, en forma de documentación, *how-tos*, tutoriales, guías, FAQs (preguntas frecuentes) e instrucciones, generalmente surgido de distintos conocimientos "blandos", en la forma de intensas interacciones entre usuarios y desarrolladores, de las reflexiones y decisiones colectivas y principalmente de la experiencia adquirida en el uso del software [4][40][56][63]

A. Canales de comunicación

Los principales canales de comunicación utilizados por la comunidad de Apache ServiceMix se basan, principalmente, en herramientas asincrónicas, a saber: listas de correo, sistema de control de versiones, WIKI (sitio web colaborativo) y *Bug Tracker* (sistema de administración de incidencias). El diseño de estas plataformas de colaboración construidas sobre internet proveen el marco en donde el conocimiento generado y adquirido es "centralizado" y funcionan como el principal recurso donde los participantes, nuevos y activos, pueden consultar, colaborar y aprender.

Igualmente es necesario mencionar la importancia del uso de herramientas sincrónicas en los casos que se requiera la resolución de problemas particulares y urgentes. Sin embargo, a los fines de establecer una base de conocimiento que pueda ser consultada y accedida posteriormente, este estilo de interacción no es útil.

Aunque todas estas herramientas colaborativas son útiles en la tarea para la cual fueron implementadas, y necesarias para dar soporte al crecimiento del proyecto, su utilización conjunta, y en general, su integración e interoperabilidad en una plataforma colaborativa única, plantea numerosos desafíos, principalmente debido a la incompatibilidad de formatos de datos entre ellas.

B. Mi experiencia como nuevo usuario

1. Unirse a la comunidad

Waterson[64] sostiene que el desarrollo de software es una actividad de conocimiento intensiva que generalmente requiere altos niveles de conocimiento del dominio, experiencia y aprendizaje intensivo de aquellos que contribuyen al desarrollo. A medida que el proyecto avanza, y la tecnología usada se complejiza, solo algunas personas que han estado involucradas activamente en el desarrollo por algún tiempo tienen la capacidad de entender la arquitectura del software y contribuir con código al desarrollo.

Aquellos que quisimos unirnos a la comunidad del proyecto tuvimos que atravesar un período de aprendizaje para poder contribuir posteriormente a partir de la experiencia adquirida, en algunos casos, aportando o sugiriendo código

Es importante que los usuarios activos y principales desarrolladores hayan

estructurado y organizado la documentación y demás recursos que les faciliten a los nuevos usuarios el comienzo en el uso de la herramienta.

En el proyecto Apache ServiceMix el avance del desarrollo, la existencia de documentación en la forma de guías, tutoriales para principiantes y algunos ejemplos básicos, me permitieron que el estudio, análisis, uso e implementación de servicios basados en Apache ServiceMix pueda ser realizado sin requerir altos niveles de conocimiento.

2. Interactuar con los demás miembros

Principalmente mediante las listas de correo, los nuevos participantes interactuamos con miembros más experimentados, incluidos los principales desarrolladores, de manera que el nivel de información intercambiada es alto.

Todos en la lista nos involucramos en la resolución de los problemas surgidos de distintas maneras: los nuevos participantes podemos realizar diferentes tipos de *testing* sobre el modelo de problema simulado, analizar reportes de *bugs*, traducir documentación o bien, simplemente seguir y entender el desarrollo de la resolución de los problemas.

Aquellos desarrolladores más experimentados o usuarios avanzados son los que ultiman los detalles de los *patchs*[44] enviados por otros miembros y los "mezclan" con la línea central de desarrollo, evitando de esta manera generar inconsistencias en el proyecto.

3. Participación en el proyecto

En la medida que los nuevos usuarios vayamos "devolviendo" algo a la comunidad, nuestra participación se va legitimando ante los demás miembros, además de eventualmente ayudar a otros miembros a resolver o encontrar nuevas vías de resolución de errores. Esa "devolución" generalmente es en la forma de reporte de *bugs*[45], código de casos de usos, pedido de nuevas características, resolución de problemas con ciertas versiones de librerías de terceros, entre otros.

Al principio la conexión de los nuevos usuarios con la comunidad es un poco débil e informal, limitándose a realizar preguntas que no puedan ser resueltas a partir de una búsqueda en los recursos publicados por el proyecto, o en otros buscadores.

Luego esta conexión se afianza y transforma en relaciones más estrechas con otros participantes, se adquieren nuevas habilidades y conocimiento que puede ser útil para los usuarios que recién comienzan.

La legitimación dentro de la comunidad se obtiene en base al nivel de participación, habilidades e involucramiento con el proyecto. Es un sistema basado únicamente en los **méritos** obtenidos [40]

La FIGURA 6.1 muestra una visión parcial de la jerarquía de usuarios

---

44 **Patch**: código corrector

45 **Bug**: fallo o deficiencia durante el proceso de creación de software

involucrados en proyectos FLOSS

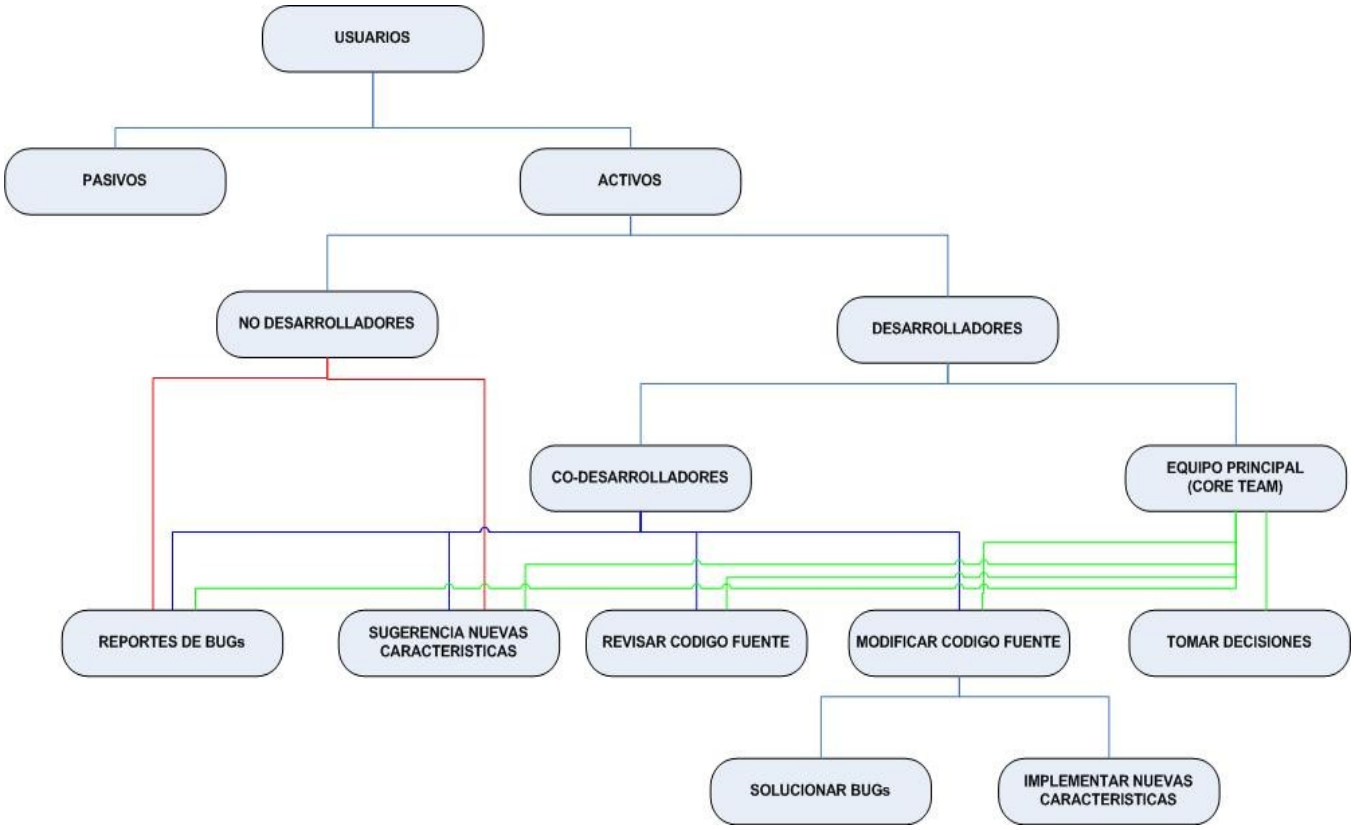


FIGURA 6.1 - Jerarquía parcial de usuarios en proyectos FLOSS (adaptada de [40])

Capítulo 7

jESBihca: Análisis

7.1. Introducción

Este capítulo pretende reunir los conceptos relacionados con integración de aplicaciones heterogéneas y específicamente sobre ciertas tecnologías de integración, en la forma de un **caso de estudio** que hace uso de *jESBihca*, el prototipo de una plataforma de integración, y que permita comprender su utilización en la resolución de una problemática específica.

El caso de estudio se refiere a la integración **de los resultados de la búsqueda de información dispersa en un conjunto específico de herramientas colaborativas asincrónicas utilizadas para la coordinación del trabajo colaborativo por parte de la comunidad de usuarios y desarrolladores de un hipotético proyecto FLOSS, el cual denominaremos *jIntegra***

Se realiza una implementación de un ESB, basado en la especificación JBI, múltiples servicios y componentes específicos de este caso de estudio, adaptadores a las distintas herramientas colaborativas y un motor de búsqueda *full text* que permita realizar búsquedas y una eficiente indexación de las diversas fuentes de información registradas. A esta solución de integración integral la he denominado ***jESBihca*** (**J**ava **ESB** para la **I**ntegración de **H**erramientas **C**olaborativas **A**sincrónicas)

La solución implementada por el autor de esta tesis pretende mostrar desde una perspectiva práctica algunos de los conceptos teóricos analizados en los capítulos incluídos en la parte I.

7.2. Herramientas Colaborativas Asincrónicas

Las herramientas colaborativas asincrónicas utilizadas por el hipotético proyecto FLOSS *jIntegra* comprenden:

A. MediaWiki: WIKI
B. Mantis BT: Bug Tracker
C. Subversion: Sistema de control de versiones
D. Sistema de archivos compartido (FTP, carpetas compartidas, File server)

A. MediaWiki: WIKI

Wikipedia[46] define un wiki (o una wiki) (del hawaiano wiki wiki, "rápido") como

> *"un sitio web colaborativo que puede ser editado por varios usuarios. Los usuarios de una wiki pueden así crear, editar, borrar o modificar el contenido de una página web, de forma interactiva, fácil y rápida; dichas facilidades hacen de la wiki una herramienta efectiva para la escritura colaborativa.*
>
> *La tecnología wiki permite que páginas web alojadas en un servidor público (las páginas wiki) sean escritas de forma colaborativa a través de un navegador, utilizando una notación sencilla para dar formato, crear enlaces, conservando un historial de cambios que permite recuperar fácilmente cualquier estado anterior de la página. Cuando alguien edita una página wiki, sus cambios aparecen inmediatamente en la web, sin pasar por ningún tipo de revisión previa"*

B. Mantis BT: Bug Tracker

Wikipedia[47] define un sistema de seguimiento de errores o incidencias como

> *" ...una aplicación de software diseñada para ayudar a los programadores a mantener un registro de los errores de software reportados"*

46 **WIKI:** es.wikipedia.org/wiki/Wiki

47 **Bug Tracker:** es.wikipedia.org/wiki/Bugtracker

C. Subversion: Control de Versiones del Código Fuente

Wikipedia[48] define SCM como

> *“ ...una aplicación informática que implementa un sistema de control de versiones: mantiene el registro de todo el trabajo y los cambios en los ficheros (código fuente principalmente) que forman un proyecto y permite que distintos desarrolladores (potencialmente situados a gran distancia) colaboren”*

D. Sistema de archivos:

Los sistemas de archivos, según Wikipedia[49],

> *“ ...estructuran la información guardada en una unidad de almacenamiento (normalmente un disco duro) de una computadora, que luego será representada ya sea textual o gráficamente utilizando un gestor de archivos. La mayoría de los sistemas operativos poseen su propio sistema de archivos”*

En el caso de estudio el sistema de archivos puede ser utilizado para distintos propósitos, por ejemplo, como soporte de un servidor de archivos, carpetas compartidas, o bien el acceso y escritura vía FTP

En la problemática planteada por el caso de estudio, estas herramientas son utilizadas de manera aislada, posiblemente desplegadas en servidores geográficamente distantes, utilizan formatos de datos propios, no compatibles entre sí, y ejecutan sobre diferentes plataformas de software/hardware

La tarea consiste en construir un sistema estándar y basado en mensajería que permita la integración ***de los resultados de la búsqueda federalizada realizada por el cliente en las diferentes fuentes de información*** enumeradas anteriormente, y retornarlos con un **formato de presentación homogéneo**, y una referencia a la herramienta de donde proviene cada resultado en particular.

Para lograr la **integración** de las distintas **herramientas** con el sistema de mensajería, es necesario que cada herramienta colaborativa tenga asociado un servicio que funcione como un **adaptador** y permita la recepción, transformación

48 **SCM:** es.wikipedia.org/wiki/Sistema_de_control_de_versión

49 **Sistema de Archivos:** es.wikipedia.org/wiki/Sistema_de_archivos

y envío de mensajes.

El sistema de mensajería debe proveer variedad de conectores a protocolos diversos, que permita cierta flexibilidad a la hora de conectar distintos tipos de clientes con el motor de búsqueda, que se comunicará a su vez con los servicios adaptadores de las herramientas colaborativas.

Serán necesarios ciertos componentes de infraestructura que brinden soporte al sistema de mensajería en determinadas cuestiones.

Los canales de mensajería facilitarán la circulación de mensajes normalizados entre los diferentes componentes internos al sistema de mensajería

El router de mensajes permitirá decidir el camino a seguir por los mensajes en base a un conjunto de reglas preconfiguradas, y fácilmente adaptables.

El transformador de mensajes facilitará la transformación entre formatos de datos, en base a un conjunto de plantillas XSL preconfiguradas y fácilmente adaptables.

Aunque la implementación del caso de estudio ofrece puntos abstractos que facilitan la extensión de la funcionalidad provista por *jESBihca*, el principal objetivo es mostrar los aspectos prácticos y más relevantes de la integración de aplicaciones, particularmente aquellos descriptos en los capítulos precedentes.

7.3. Escenarios analizados

Para realizar la búsqueda en las distintas fuentes de información registradas el cliente envía el texto que desea buscar al sistema de mensajería a través de determinado protocolo.

Existen diferentes alternativas de búsqueda:

- El cliente conoce exactamente que buscar con lo cual ingresa el texto encerrado entre comillas dobles.
- El cliente conoce parcialmente el texto a buscar. Puede utilizar caracteres *wildcard (*"*", "?"*)*
- El cliente conoce palabras sueltas, y posiblemente no relacionadas entre si, del texto a buscar. Puede utilizar operadores lógicos (*AND, OR*) para realizar la búsqueda

Se implementaron **tres escenarios** para realizar las búsquedas a través de *jESBihca*

7.3.1. Escenario 1: Búsquedas basadas en texto plano

Este escenario típicamente involucra la participación de un humano que a través de una interface web realiza una búsqueda basada en texto plano a *jESBihca* quien indagará por la información buscada en todas las fuentes de información registradas. Luego devolverá un conjunto de resultados con formato homogéneo, con detalles del recurso encontrado, relevancia, autor, resumen, y una referencia a la fuente de información de donde surge el resultado, para poder acceder físicamente el recurso encontrado, en caso que sea posible.

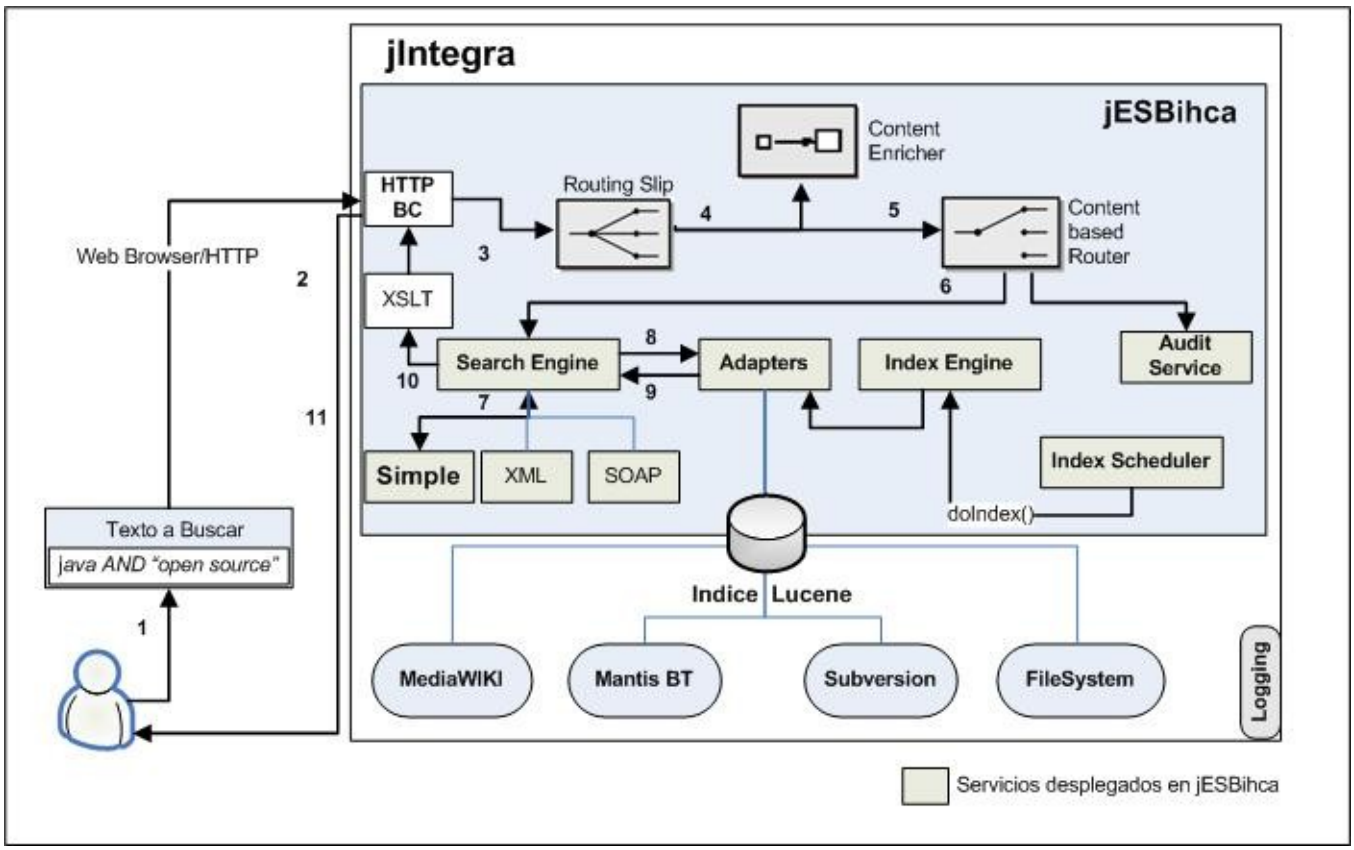


FIGURA 7.1 - Diseño de alto nivel del escenario de búsqueda basado en texto plano

7.3.2. Escenario 2: Búsquedas basadas en documentos XML

En este escenario el cliente es típicamente una aplicación que envía el texto a buscar, junto a otros parámetros, encapsulado en un documento XML con un **formato preacordado**. El servicio XML desplegado en *jESBihca* y escuchando por los requerimientos se encarga de procesar el documento entrante, invocar al motor de búsqueda y encapsular los resultados encontrados en un documento XML de salida, de acuerdo a un formato ya establecido. Este documento es la respuesta que se envía a la aplicación cliente que inicio la interacción.

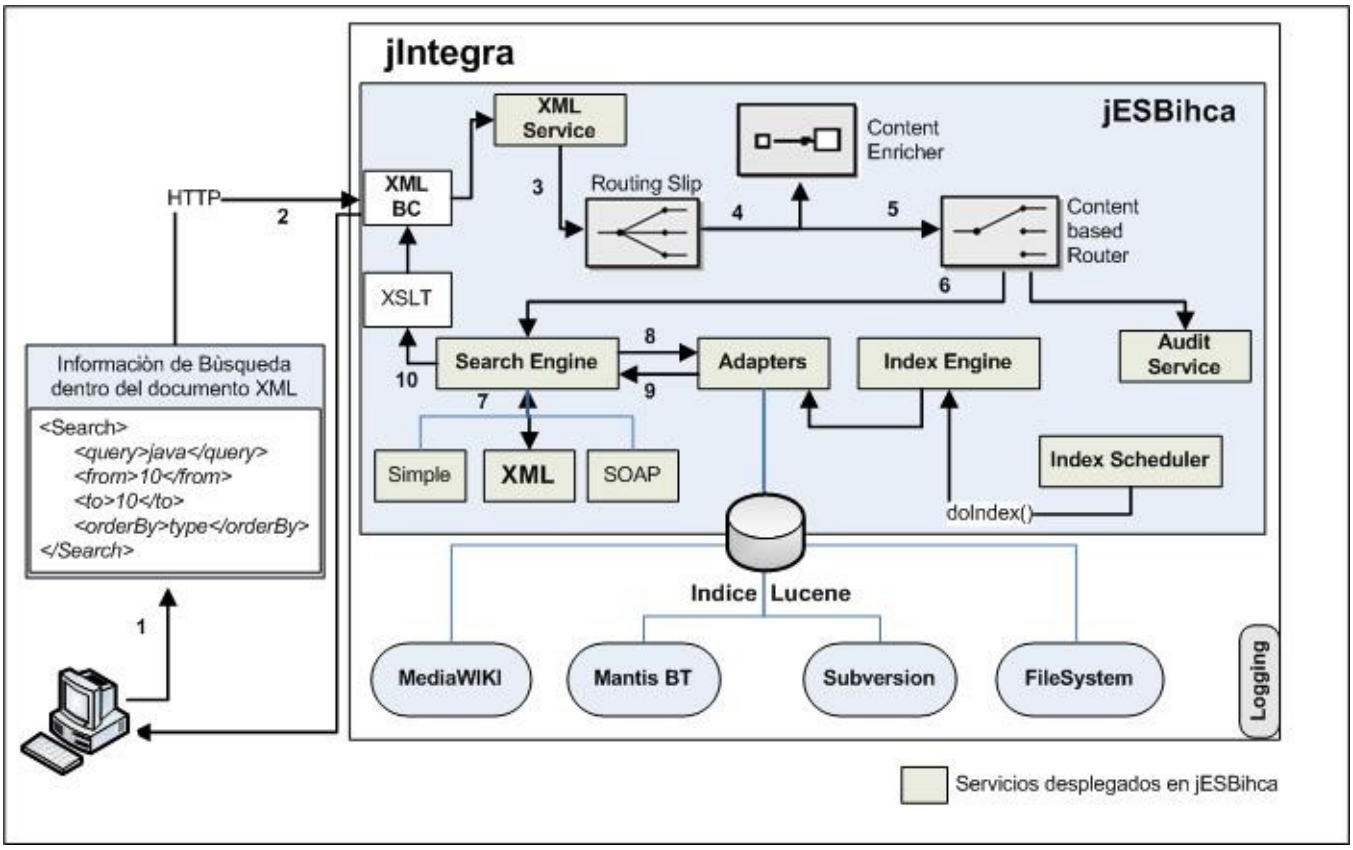


FIGURA 7.2 - Diseño de alto nivel del escenario de búsqueda basado en XML

7.3.3. Escenario 3: Búsquedas basadas en mensajes SOAP

En este escenario el cliente es una aplicación que envía el texto a buscar, junto a otros parámetros, a través de un mensaje SOAP.

En el caso de estudio analizado los requerimientos SOAP utilizan HTTP como transporte; por lo tanto el pedido es atendido por un servicio HTTP desplegado en *jESBihca* que se encarga de delegar el control en un servicio SOAP.

Este último obtiene del mensaje SOAP los datos que necesita el motor de búsqueda para recuperar la información dispersa en las diferentes fuentes de información registradas. El resultado es encapsulado en un mensaje SOAP y es enviado como respuesta al cliente.

La estructura del mensaje SOAP enviado por la aplicación del servicio SOAP debe ajustarse al documento WSDL publicado por ***jESBihca.*** Este documento es desplegado en una URL de acceso público.

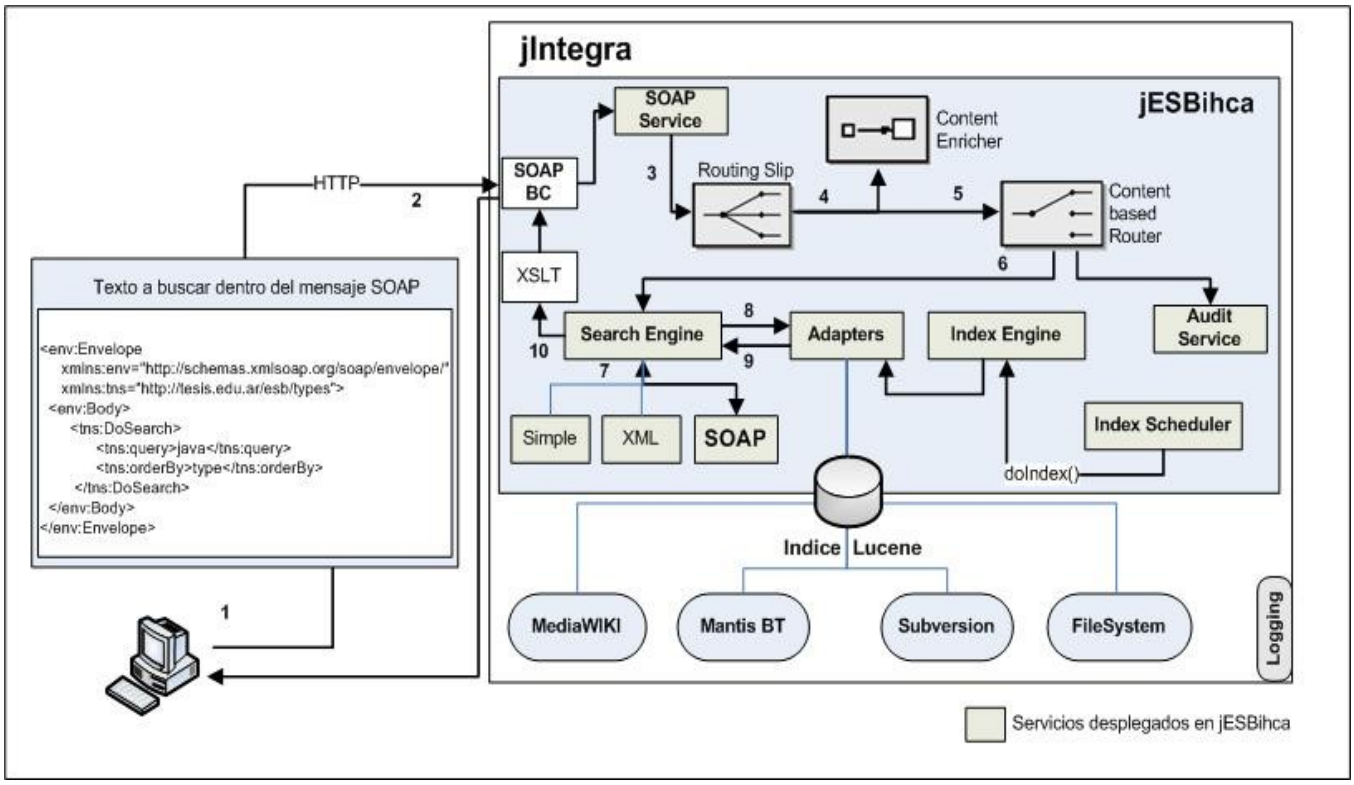


FIGURA 7.3 - Diseño de alto nivel del escenario de búsqueda basado en SOAP

Capítulo 8

jESBihca: Implementación

8.1. Introducción

En este capítulo se comienza a analizar la implementación de la plataforma ESB denominada *jESBihca*.

En primer lugar se describirán los conceptos relacionados con esta solución, las tecnologías involucradas y arquitectura en general, y posteriormente se describirán brevemente las herramientas utilizadas para su construcción.

Por último, se analizarán los detalles técnicos internos de la implementación, ilustrados con capturas de pantalla de porciones de código fuente, de archivos de configuración y diagramas explicativos del caso de estudio analizado en el capítulo precedente.

8.2. Generalidades

Un concepto importante presentado durante el análisis de la problemática que dio origen a este trabajo, fue el **uso de FLOSS** para la integración de aplicaciones heterogéneas

El haber considerado el uso y adaptación FLOSS para la implementación del caso de estudio tiene, principalmente, **dos razones**.

En primer lugar mi **fuerte convicción** sobre las ventajas tecnológicas y sociales que FLOSS tiene respecto al software privativo (de "código fuente cerrado" y licencias de uso restrictivas).

En segundo término, el **ser consciente** sobre los **limitados conocimientos, recursos técnicos y tiempo** que tenía para poder realizar la **implementación de un ESB** *(Enterprise Service Bus)* que cumpliera todos los aspectos y capacidades enumeradas en los capítulos de la parte 1, y principalmente que se adaptara a algún estándar abierto y actual de la industria del software. No hubiese sido una tarea fácil para mí, e incluso siquiera posible, el analizar, diseñar, implementar y probar un ESB construído "desde cero", y sin el uso de FLOSS.

Para la elección de las herramientas utilizadas en la implementación de *jESBihca* tuve preferencia respecto a aquellas que tuviesen licencias libres, que me permitieran estudiar, modificar, adaptar o bien optimizar el funcionamiento originalmente provisto, ésto gracias a la disponibilidad del código fuente, y que no tuvieran costo de licencia para su uso, adaptación, optimización y redistribución

La investigación y búsqueda de las herramientas más adecuadas para la implementación de *jESBihca* no fue exhaustiva. Solamente fueron estudiadas aquellas herramientas que cumplieran con los requisitos descriptos en el párrafo anterior. Por lo tanto no fue realizado un análisis comparativo respecto a herramientas similares pero que fallasen en el cumplimiento de alguno de los requisitos mencionados.

8.3. Objetivos de la solución implementada

Entre los principales objetivos que pretende alcanzar el desarrollo del framework ESB, se encuentran:

- ✔ Proveer un modelo simple para la implementación de soluciones de integración de mediana complejidad
- ✔ Ofrecer un entorno que posibilite comportamiento asincrónico y orientado a mensajes.
- ✔ Que sea una solución robusta, escalable, fácilmente extensible y que pueda ejecutar sobre distintas plataformas de software/hardware.

Teniendo en cuenta los objetivos planteados fue necesario seleccionar las herramientas que faciliten el desarrollo, *testing* y mantenimiento del framework.

Era necesario que los componentes implementados fueran débilmente acoplados para facilitar la modularización del framework; se requería que exista una clara separación entre la lógica de integración y de negocios propia de los escenarios del caso de estudio a implementar y que existan puntos de extensión lo

suficientemente abstractos para abarcar la mayor cantidad de escenarios posibles, pero, a su vez, que contengan la suficiente lógica para promover la reusabilidad y portabilidad.

8.4. Tecnologías

La elección de las tecnologías de desarrollo para la construcción de un SOA usando un ESB no es tan crítico como puede serlo en el desarrollo tradicional de aplicaciones, ya que se espera que los servicios y clientes puedan interoperar entre sí, a pesar de estar codificados o ejecutar en plataformas de hardware/software diferentes.

Actualmente las 2 principales opciones en tecnologías de desarrollo son JEE y .NET. Según el objetivo de esta tesis la plataforma, frameworks y herramientas de desarrollo tienen que tener licencias libres u open source, a fin de poder reutilizar porciones de código, adaptar, optimizar y estudiar su funcionamiento. Estas precondiciones restringen la elección a herramientas libres u open source basadas en JEE.

Fueron seleccionadas numerosas herramientas libres, muchas de las cuales no fueron utilizadas en su totalidad, sino solo la funcionalidad requerida por jESBihca. Se **priorizaron** herramientas y tecnologías modernas, basadas en **Java**, y preferentemente surgidas o apoyadas desde la fundación **Apache**[50], haciendo hincapié en aquellas tecnologías que provean funcionalidad escalable, que sean fácilmente configurables y principalmente extensibles y adaptables a diferentes escenarios.

Sin embargo, aunque todas ellas sean importantes y necesarias para la implementación, una de ellas, Apache ServiceMix[66], constituye el cimiento sobre el cual se construyó jESBihca, ya que permitió que la solución sea compatible con JBI[54], estándar promovido por la JCP[51], aceptado mundialmente por los principales proveedores IT, y que permite la construcción de software independiente del proveedor.

8.5. Arquitectura

La construcción de aplicaciones usando SOA difiere significativamente de técnicas usadas en otras arquitecturas de software.

SOA requiere que el desarrollador de software construya servicios (o integre funciones de negocios existentes como servicios) que sean reusables en diferentes aplicaciones. Cada servicio es creado pensando en la reusabilidad, que se obtiene al desacoplar los servicios consumidores de los servicios proveedores.

---

50 **Fundación Apache**: http://apache.org

51 **JCP:** Java Community Process - http://jcp.org

JBI provee las bases para la construcción de un SOA. Posibilita la integración de funciones de negocios existentes como servicios y desacoplar los servicios consumidores de los servicios proveedores. Provee soporte directo para la composición de aplicaciones a través del uso de paquetes de servicios (en terminología JBI, denominados *Services Assemblies*), que permiten componer aplicaciones a partir de las interfaces de entidades denominadas *Services Units*.

Este soporte directo para la construcción de aplicaciones compuestas y el proveer de una infraestructura estándar de mensajería hizo de JBI el candidato ideal para construir la infraestructura que permita el desarrollo de aplicaciones orientadas a servicios y la integración orientada a servicios de aplicaciones existentes.

Por lo anterior es que **jESBihca** utiliza una implementación JBI, **Apache ServiceMix**, para constituirse como una plataforma de integración de acuerdo a los principios estudiados de SOA, que provea una arquitectura estándar y abierta basada en servicios, de fácil despliegue, configuración y administración

La FIGURA 8.1 muestra gráficamente una versión simplificada de la arquitectura general de la solución de integración del caso de estudio planteado

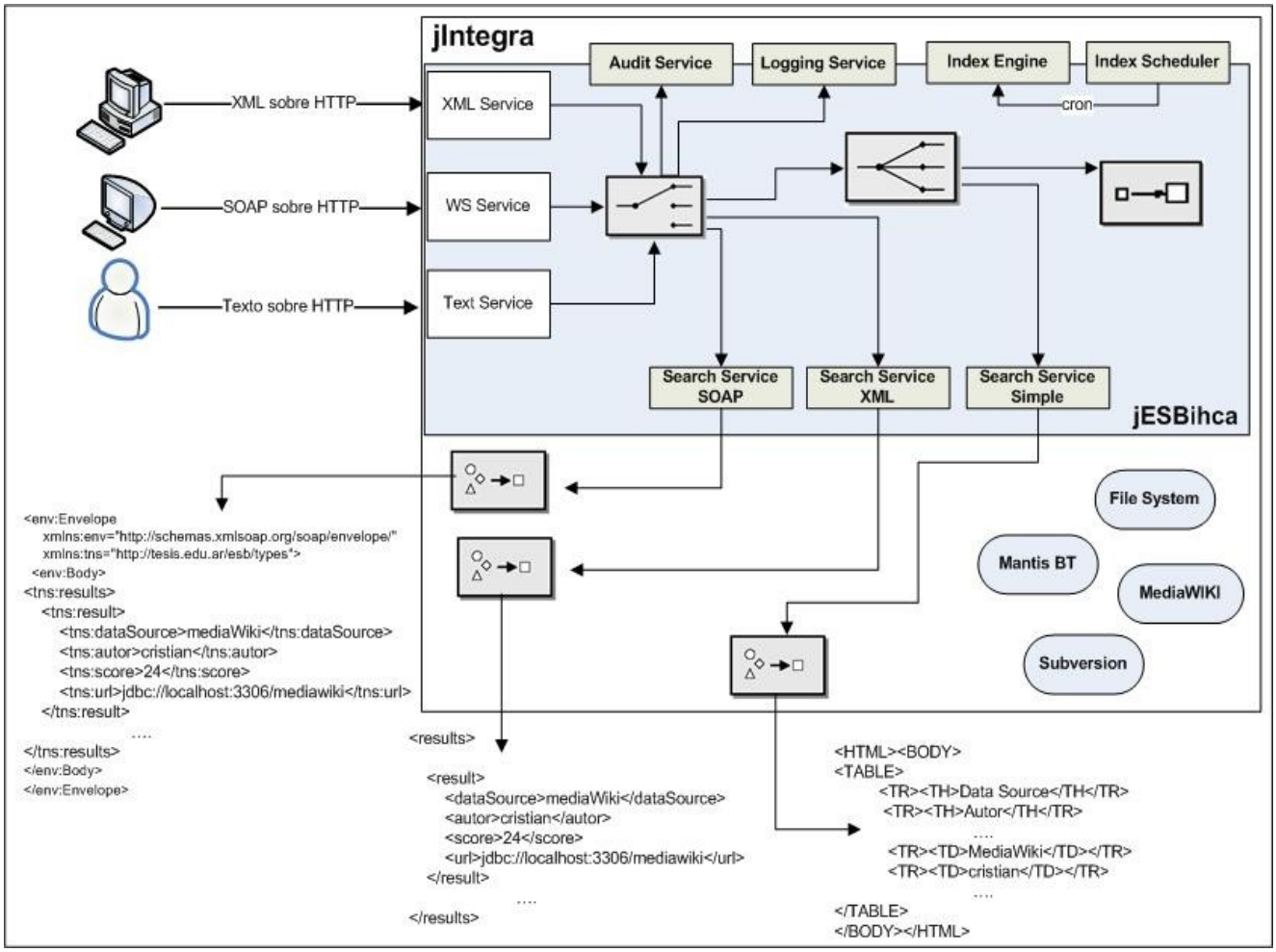


FIGURA 8.1 - Arquitectura General del Caso de Estudio presentado

8.6. Frameworks y Librerías usadas

A. Spring Framework

A.1. Arquitectura

Arquitectónicamente (FIGURA 8.2) el framework Spring está organizado en seis grandes módulos

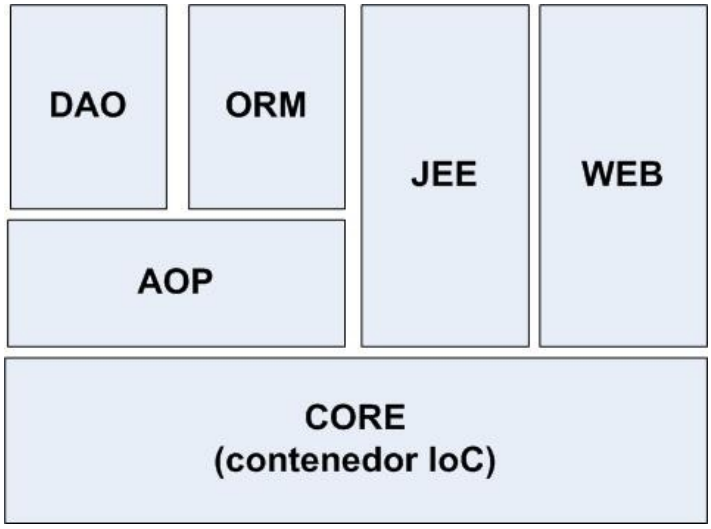


FIGURA 8.2 – Arquitectura del framework Spring

Spring es un framework "liviano" y open source que cubre prácticamente todos los aspectos de la implementación de una aplicación empresarial típica [72]

Sus características principales y diferenciales incluyen:

- ***Lightweight*:** es un framework liviano tanto en tamaño (el framework puede ser distribuido en un archivo *jar* que ocupa aproximadamente 2.6 mb), como en *overhead,* el cual es insignificante respecto a aplicaciones que no utilicen el framework.
- **No es intrusivo**: los objetos en una aplicación basada en Spring usualmente no tienen dependencias con clases de Spring.
- **DI (*Dependency Injection*):** Spring promueve el débil acoplamiento mediante una técnica conocida como DI. Cuando DI es utilizado los objetos obtienen "pasivamente" sus dependencias en lugar de buscar y crearlas por si mismos. Es el contenedor quien instancia y crea las dependencias entre los objetos sin esperar que sea "llamado".
- **Orientado a Aspectos:** Spring ofrece un soporte amplio para la

programación orientada a aspectos (AOP) que facilita el desarrollo de aplicaciones separando la lógica especifica de negocios de los servicios del sistema (auditoria, *logging*, transacciones)

- **Contenedor**: Spring es considerado un contenedor en el sentido de contener y administrar el ciclo de vida y configuración de los objetos de la aplicación. De esta manera, en ambientes Spring es posible **declarar** como cada objeto de la aplicación debería ser creado, configurado y asociado con otros objetos.
- **Framework:** Spring permite configurar y componer aplicaciones complejas a partir de componentes simples. Los objetos de la aplicación son compuestos **declarativamente**, usualmente en archivos XML.
- **Funcionalidad de Infraestructura**: Spring provee mucha funcionalidad propia de la infraestructura (transacción, integración, persistencia), generalmente del estilo de *templates*, lo cual permite que el desarrollador centralice sus esfuerzos en la lógica de negocios propia del dominio del problema.
- **Buenas prácticas de desarrollo**: Promueve prácticas de codificación reusable y modular

A.1.1. CORE – Contenedor IoC

La interface *org.springframework.beans.factory.BeanFactory* es la representación del contenedor IoC de Spring. Es responsable de instanciar, configurar y resolver las dependencias entre los objetos de la aplicación Aunque es común el termino IoC (Inversion of Control), se lo está reemplazando por DI (*Dependency Injection*) que refleja mejor cual es su función

Existen numerosas implementaciones de BeanFactory, siendo las más utilizada XmlBeanFactory que permite modelar en XML los objetos de negocios y las interdependencias entre ellos.

Este documento XML es la metadata que junto a los POJOs permite al contenedor crear un sistema totalmente configurado y listo para ser utilizado.

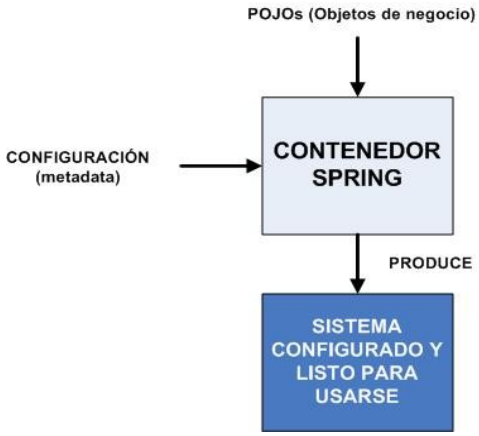


FIGURA 8.3 – Contenedor IoC de Spring

Aunque la configuración escrita en XML es el medio más utilizado, no es el único El contenedor esta totalmente desacoplado del formato en el cual esta metadata es escrita.

La definición de *beans* dentro del archivo de configuración escrito en XML se realiza en elementos **<bean>** dentro de un elemento de nivel superior **<beans>**

Estos *beans* corresponden a los objetos que componen la aplicación. Incluye definiciones de *beans* para objetos de la capa de servicios, objetos de acceso a datos (DAO), objetos de presentación, objetos de infraestructura (*queues* o *topics* JMS, SessionFactory de Hibernate), entre otros

La FIGURA 8.4 muestra uno de los archivos XML que permite configurar el servicio de indexación implementado en el caso de estudio. Este archivo utiliza "sintaxis" Spring

```
<import resource="classpath:esbcore-applicationContext-hibernate.xml" />

<bean id="indexer" class="ar.edu.tesis.indexing.ESBIndexerImpl">
     <property name="adapters"><ref local="esbAdapters" /></property>
</bean>

<bean id="svnAdapter" class="ar.edu.tesis.adapters.impl.SVNAdapterImpl">
    <property name="adapterConfig"><ref bean="adapterConfig"/></property>
    <property name="name"><value>svnAdapter</value></property>
    <property name="projectName"><value>testrepo</value></property>
    <property name="repoURL"><value>file:///home/cristian/myrepo</value></property>
</bean>

 <bean id="hibernateAdapter" class="ar.edu.tesis.adapters.impl.HibernateAdapterImpl">
    <property name="name"><value>hibernateAdapter</value></property>
    <property name="sessionFactory"><ref bean="sessionFactory" /></property>
 </bean>

 <bean id="esbAdapters" class="ar.edu.tesis.core.ESBAdapter" init-method="start" destroy-method="stop">
     <property name="adapterConfig"><ref bean="adapterConfig" /></property>
     <property name="adapters">

     <list>
         <ref local="jdbcAdapter" />
         <ref local="svnAdapter" />
         <ref local="hibernateAdapter" />
         <bean class="org.compass.spring.device.SpringSyncTransactionGpsDeviceWrapper">
             <property name="adapter" ref="hibernateAdapter" />
         </bean>
     </list>
   </property>
 </bean>
```

FIGURA 8.4 – Extracto de configuración del servicio de indexación del caso de estudio

A.1.2. AOP

Este módulo sirve como base para la definición de aspectos propios de la

aplicación. Tal como lo hace DI (*Dependency Injection*), AOP[52] facilita el débil acoplamiento de los objetos de la aplicación.

A.1.3. DAO[53]

Este módulo permite abstraer las complejidades y tareas repetitivas que hacen a los accesos vía JDBC (*Java DataBase Connectivity*) a base de datos relacionales. Construye también una capa de excepciones más explicativas que los mensajes de error dados por los distintos servidores de base de datos.

A.1.4. ORM[54]

Este módulo facilita la creación de DAOs para la mayoría de las conocidas soluciones ORM existentes. En el caso de estudio implementado se hace uso de las facilidades que brinda Spring para integrarse con el ORM Hibernate[55]

A.1.5. JEE

Este módulo ofrece fácil integración y *templates* para distintas extensiones JEE, tales como JMX (*Java Management eXtensions*), JCA (*J2EE Connector Architecture*), JMS (*Java Message Service*)

A.1.6. Web

El módulo web facilita la codificación de funcionalidad requerida por las aplicaciones web, tales como *upload* de archivos, *bindings* de parámetros desde el *request,* entre otras.

Spring también ofrece una implementación propia del patrón de diseño MVC (Model View Controller), además de ofrecer puntos de integración con otros frameworks conocidos que implementan MVC, tal es el caso de Apache Struts, Webworks, entre otros. De esta manera también promueve técnicas de débil acoplamiento en la capa web de la aplicación

A.2. AOP y DI (*Dependency Injection*)

Estos dos aspectos implementados en Spring son fundamentales para lograr el débil acoplamiento entre los objetos de la aplicación

---

52 **Aspect-oriented Programming**: es.wikipedia.org/wiki/Programación_Orientada_a_Aspectos

53 **Patrón DAO (Data Access Object)**: es.wikipedia.org/wiki/Data_Access_Object

54 **ORM**: Object-relational Mapping

55 **Hibernate**: http://hibernate.org

En cualquier aplicación tradicional donde existan más de 2 objetos, cada objeto es responsable de obtener las referencias a los objetos con los cuales quiere colaborar (sus dependencias). Esto conduce a código altamente acoplado y dificultoso para realizar *testing.*

Al aplicar DI los objetos reciben sus dependencias en el momento de creación por alguna entidad externa que realiza la coordinación de los objetos de la aplicación.

De esta manera las **dependencias** son "**inyectadas"** dentro de los objetos (FIGURA 8.5). DI significa entonces una inversión en la responsabilidad con respecto a como los objetos obtienen las referencias a sus colaboradores.

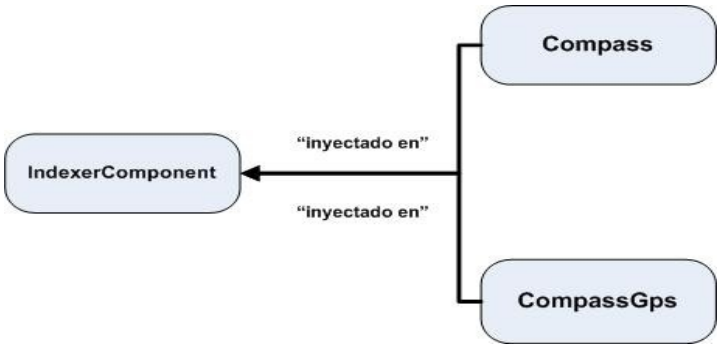


FIGURA 8.5 – "Inyección" de dependencias usando Spring

La FIGURA 8.6 muestra un extracto de un archivo de configuración XML donde se implementa este concepto, mientras que la FIGURA 8.7 es el extracto de código Java correspondiente a la configuración anterior

```
<bean id="indexer" class="ar.edu.tesis.indexing.ESBIndexerImpl">
    <property name="adapters"><ref local="esbAdapters" /></property>
</bean>
```

*Se setea la propiedad **adapters** del bean ar.edu.tesis.indexing.ESBIndexerImpl mediante **DI (Dependency Injection)***

FIGURA 8.6 – Extracto de la configuración del concepto DI en Spring

```
public class ESBIndexerImpl implements ESBIndexer, InitializingBean {


    private Log log = LogFactory.getLog(this.getClass());

    private boolean buildIndex = true;
    private int lazyTime = 10;
    private IndexerThread iThread;

    protected ESBAdapters adapters;

    public void afterPropertiesSet() throws Exception {
        if (buildIndex) {
            Assert.notNull(adapters, "ERROR. LOS ADAPTADORES NO FUERON SETEADOS AUN");

        }
    }

    public void doIndex() {
       iThread = new IndexerThread(adapters);
       iThread.start();
    }

    public ESBAdapters getAdapters() {
        return adapters;
    }

    public void setAdapters(ESBAdapters adapters) {
        this.adapters = adapters;
    }
```

FIGURA 8.7 – Extracto de código correspondiente a la configuración mostrada en la FIGURA 8.6

El principal beneficio de DI es el débil acoplamiento. Si los objetos conocen únicamente las interfaces de sus colaboradores, no como están implementados o fueron instanciados, luego las dependencias pueden ser cambiadas sin que se vean afectados los objetos que de ellas dependan.

Aunque DI posibilita ligar débilmente los objetos, AOP permite capturar funcionalidad que es utilizada a través de toda la aplicación en componentes reusables.

AOP es considerada una técnica que promueva la separación de conceptos en un sistema de software. Un sistema esta compuesto de múltiples componentes, cada uno responsable de una porción especifica de funcionalidad. Sin embargo, usualmente, estos componentes también tienen responsabilidades adicionales más allá de su funcionalidad específica. Tal es el caso de los servicios de *logging, seguridad, transacción*, que se encuentran implementados o referenciados en componentes cuya funcionalidad es otra.

Tales servicios de sistema comúnmente son necesarios en múltiples componentes de software, lo cual provoca, básicamente, dos inconvenientes:

- El código que implementa esos sistemas se encuentra duplicado en diferentes componentes de software.
- Los componentes que necesiten alguna de dichas funcionalidades son

"sobrecargados" con código que no representa un aspecto de la funcionalidad especifica por la cual fueron diseñados.

La FIGURA 8.8 muestra gráficamente esta situación donde los objetos de negocios están íntimamente relacionados con estos servicios de sistema.

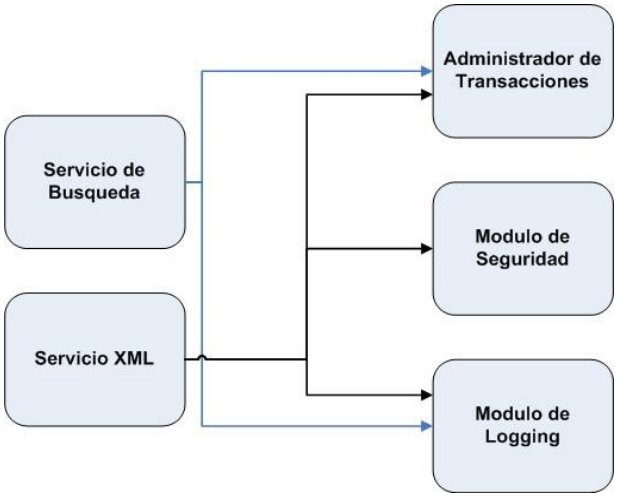


FIGURA 8.8 – Dependencias no deseadas entre objetos de negocios y servicios del sistema

AOP permite modularizar estos servicios y luego configurarlos declarativamente en los componentes que debería afectar (FIGURA 8.9). De esta manera los componentes de software "ignoran" tales servicios de sistema, y fortalecen su condición de POJOs[56]

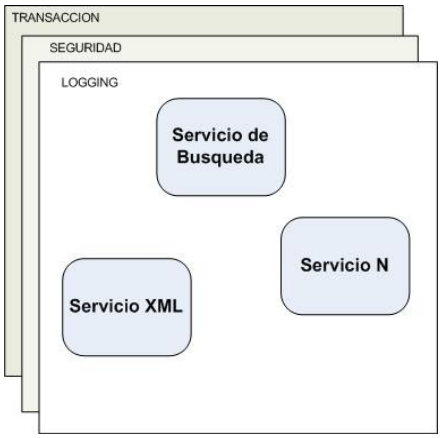


FIGURA 8.9 – Servicios transversales aplicados declarativamente a componentes de software

56 **POJO:** http://es.wikipedia.org/wiki/Plain_Old_Java_Object

B. Apache XBean

XBean es una extensión de Spring que facilita la escritura de archivos de configuración XML, principalmente al hacerlos más descriptos para los humanos.

De esta manera, XBean permite mapear elementos XML y atributos a clases JavaBean y propiedades usando *reflection* y convenciones simples [69]

En el caso de estudio implementado esta librería es utilizada con mucha frecuencia en los archivos de configuración de los servicios desplegados en el contenedor JBI.

Las figuras 8.10 y 8.11 muestran la **misma configuración** utilizando XBean (FIGURA 8.10) y Spring (FIGURA 8.11) que permiten apreciar la mayor expresividad de XBean sobre Spring

```
<beans xmlns:sm="http://servicemix.apache.org/config/1.0">

  <sm:container id="jbi" embedded="true">
    <sm:broker>
      <sm:securedBroker authorizationMap="#authorizationMap">
        <sm:flows>
          <sm:sedaFlow />
          <sm:jmsFlow jmsURL="tcp://localhost:61616" />
          <sm:jcaFlow connectionManager="#connectionManager"
                      jmsURL="tcp://localhost:61616" />
        </sm:flows>
      </sm:securedBroker>
    </sm:broker>
  </sm:container>

</beans>
```

FIGURA 8.10 – Ejemplo de configuración utilizando XBean

```
<beans>
  <bean id="jbi" class="org.apache.servicemix.jbi.container.SpringJBIContainer">
    <property name="embedded" value="true" />
    <property name="broker">
      <bean class="org.apache.servicemix.jbi.security.SecuredBroker">
        <property name="authorizationMap" ref="authorizationMap" />
        <property name="flows">
          <list>
            <bean class="org.apache.servicemix.jbi.flows.seda.SedaFlow" />
            <bean class="org.apache.servicemix.jbi.flows.jms.JMSFlow">
              <property name="jmsURL" value="tcp://localhost:61616" />
            </bean>
            <bean class="org.apache.servicemix.jbi.flows.jca.JCAFlow">
              <property name="jmsURL" value="tcp://localhost:61616" />
              <property name="connectionManager" ref="connectionManager" />
            </bean>
          </list>
        </property>
      </bean>
    </property>
  </bean>
</beans>
```

FIGURA 8.11 – Mismo ejemplo de configuración que la FIGURA 8.10, pero utilizando Spring

C. Apache Maven

Apache Maven es una herramienta para la administración de proyectos de software en Java. Basado en el concepto de POM (*project object model*) puede administrar el proceso de construcción, generación de reportes y documentación desde una única pieza de información

Permite la construcción de un proyecto utilizando su POM y una serie de *plugins* compartidos por todos los proyectos que utilizan Maven, ofreciendo de esta manera un sistema de construcción uniforme [67]

Entre las principales características de este proyecto figuran:

- Configuración simple y sencilla que permite generar la estructura de los proyectos en poco tiempo
- Potente administrador de dependencias y sus versiones. Utiliza un repositorio central de *jars* de manera que los proyectos busquen sus dependencias en ese repositorio o sean descargadas automáticamente de la web, en caso que no las encuentre.
- Permite trabajar con múltiples proyectos al mismo tiempo
- Extensible. Es posible la creación de *plugins* para tareas específicas.
- Integración con SCM (CVS/SVN)

Principios de Maven

Inspirados en la idea de Christopher Alexander[57] los principios de Maven son:

- Convención sobre configuración
- Ejecución declarativa
- Reuso de la lógica de construcción
- Organización coherente de dependencias

Convención sobre Configuración

La principal convención utilizada por Maven es la estructura estándar de directorios para los fuentes del proyecto, recursos, archivos de configuración, salida generada y documentación De esta manera, conociendo esta estructura de directorios, es fácil ubicar recursos en otros proyectos Maven.

La FIGURA 8.12 muestra gráficamente la estructura de directorios que utiliza Apache Maven por convención.

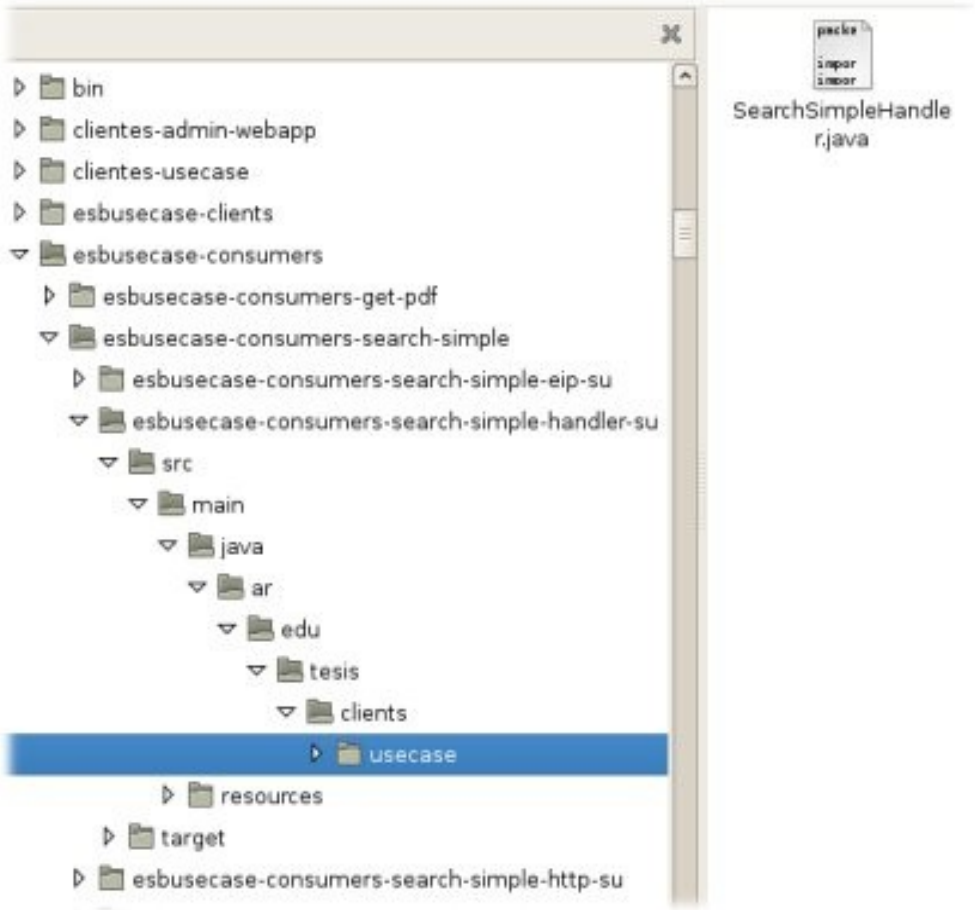


FIGURA 8.12 – Convención de la estructura de directorios usada por Apache Maven

57 **Christopher Alexander:** http://es.wikipedia.org/wiki/Christopher_Alexander

La convención sobre el nombrado estándar de directorios y salidas primarias de los proyectos Maven también es muy útil para tener una clara visión e inmediata comprensión de la ubicación, tipo y versiones de recursos.

Un ejemplo del uso de una convención de nombrado estándar se da en el nombre de las dependencias. El nombre *commons-logging-1.2.jar* nos da la información necesaria sobre esta librería, lo cual permite deducir fácilmente que se trata de la librería *commons-logging* en su versión 1.2.

Ejecución Declarativa

En Maven todo es ejecutado de manera declarativa utilizando el archivo POM y los *plugins* configurados dentro de este archivo. Sin el archivo POM Maven no es útil

La FIGURA 8.13 muestra un extracto del POM correspondiente al proyecto ***core*** del caso de estudio implementado

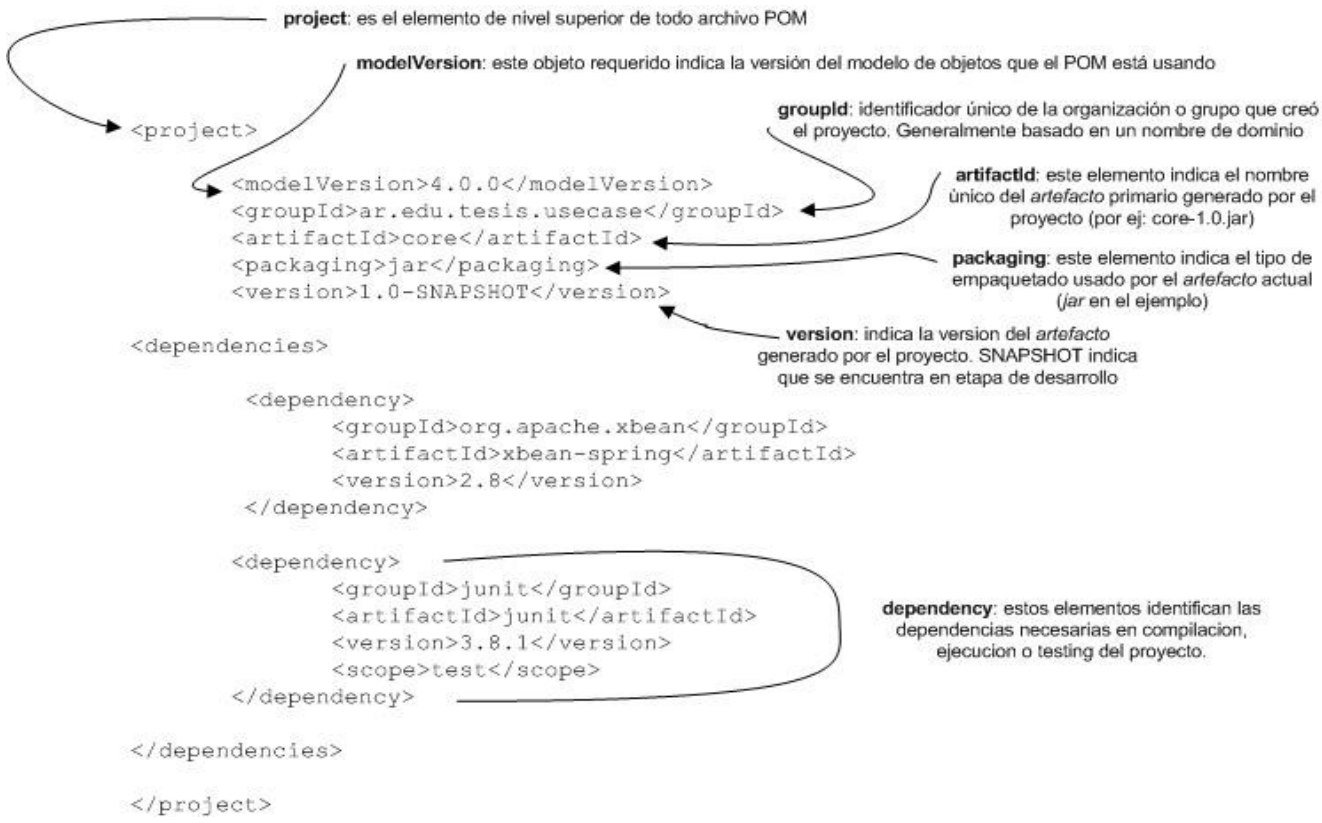


FIGURA 8.13 – Extracto de archivo POM usado por Apache Maven

Este POM permite compilar, testear y generar una documentación básica para tal proyecto.

Toda la funcionalidad implementada y basada en convenciones es realizada por

Maven a partir de un archivo POM del cual "extienden" implícitamente todos los POMs específicos de los proyectos. Lo importante de este "super POM" es que abstrae toda la funcionalidad necesaria para implementar las convenciones de Maven.

Organización Coherente de las dependencias

Como se observa en la FIGURA 8.14, la versión simplificada del archivo POM tiene dos dependencias

```
....
<dependencies>

        <dependency>
              <groupId>org.apache.xbean</groupId>
              <artifactId>xbean-spring</artifactId>
              <version>2.8</version>
        </dependency>

       <dependency>
              <groupId>junit</groupId>
              <artifactId>junit</artifactId>
              <version>3.8.1</version>
              <scope>test</scope>
       </dependency>

</dependencies>

....
```

FIGURA 8.14 - Extracto de la FIGURA 8.13

Este POM indica que el proyecto tiene 2 dependencias: una de la librería XBean, y otra de JUnit, en las versiones 2.8 y 3.8.1 respectivamente.

Una dependencia es una referencia a un librería especifica que reside en un repositorio.

Para que Maven pueda ubicar esa librería y poder satisfacer la dependencia necesita conocer en que repositorio buscar, y ciertos datos de la dependencia.

Una dependencia es identificada unívocamente a partir de los identificadores *groupId, artifactId y versión*

En el POM se le indica a Maven que dependencias espera y necesita el proyecto, pero no se especifica donde las puede ubicar físicamente

Maven toma los identificadores de dependencias declarados en el POM y los envía a su mecanismo interno de dependencias. De esta manera ya no es necesario enfocarse en un conjunto de archivos JAR, sino que se especifican dependencias lógicas.

En este caso particular, el proyecto no necesita el JAR junit-3.8.1.jar, sino que necesita la versión 3.8.1 del *proyecto (artifact)* **junit** dentro del grupo *(group)*

junit.

La FIGURA 8.15 muestra el patrón general para la disposición de las carpetas en el repositorio

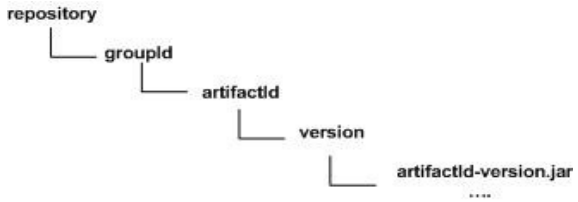


FIGURA 8.15 – Disposición de carpetas en el repositorio de Apache Maven

Cuando se instala Maven y se ejecuta por primera vez crea el repositorio local en la ubicación por defecto ~/.m2/repository

En el caso de uso implementado el *groupId* corresponde al nombre de dominio **ar.edu.tesis.**

La FIGURA 8.16 muestra parcialmente el repositorio usado para la implementación del caso de estudio, y una de las dependencias del framework del caso de estudio implementado, que tiene como *groupId a* **ar.edu.tesis.fmk**, *artifactId* a **fmk-core,** en su versión **1.0**

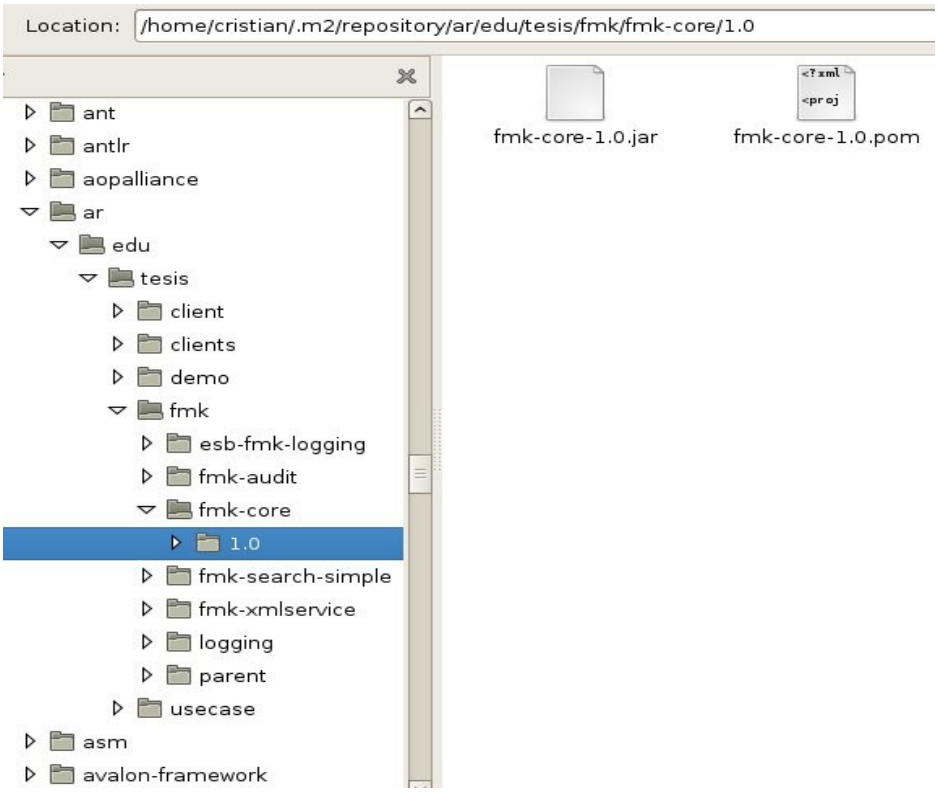


FIGURA 8.16 – Repositorio de dependencias utilizado en el caso de estudio

D. Hibernate

Hibernate es una herramienta de mapeo objeto/relacional para ambientes Java.

El término ORM (*Object-relational Mapping*) se refiere a la técnica de realizar mapeos entre representaciones de datos en el modelo de objetos a un modelo de datos relacional con un esquema basado en SQL [73]

Hibernate no sólo se ocupa del mapeo básico entre clases Java y tablas de la base de datos, y conversión entre tipos de datos Java a tipos de datos SQL, sino que provee un mecanismo avanzado de recuperación y consultas de datos que facilita y reduce el tiempo de desarrollo.

La FIGURA 8.17 muestra gráficamente la arquitectura de ambientes donde se utilice Hibernate como ORM

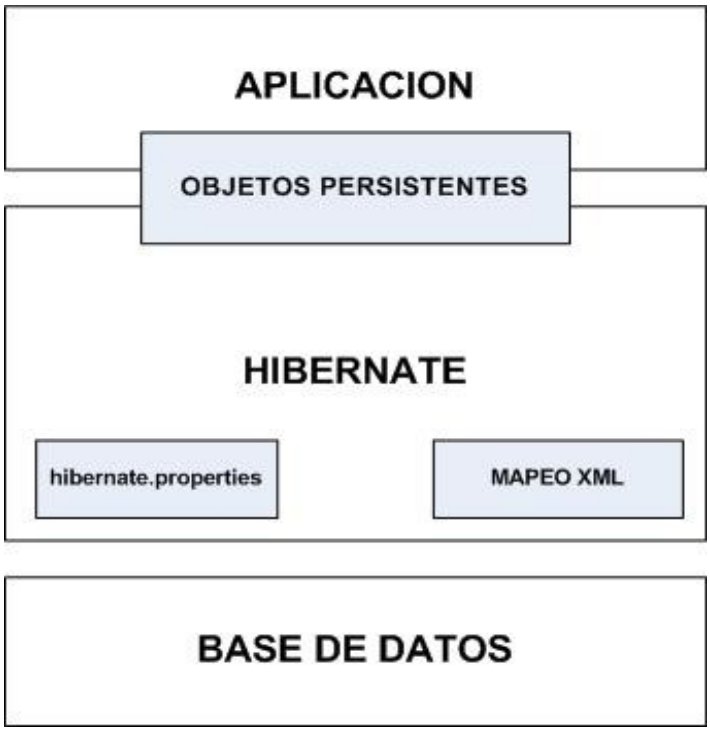


FIGURA 8.17 – Arquitectura de ambientes que utilizan Hibernate como ORM

Por otra parte, la FIGURA 8.18 muestra la arquitectura con mayor detalle de los principales componentes

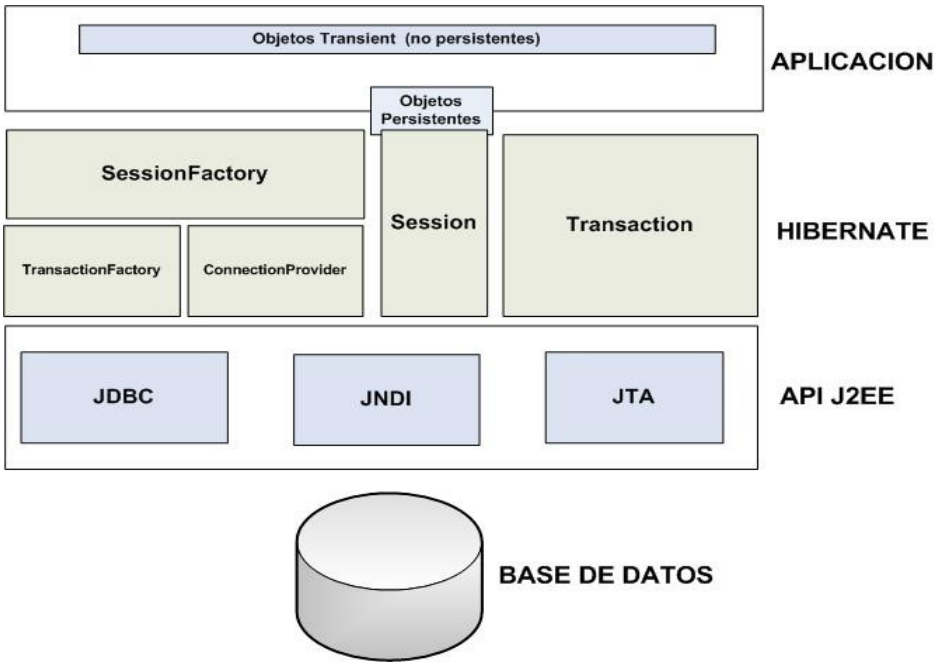


FIGURA 8.18 – Ubicación de Hibernate en las capas de una arquitectura típica

La FIGURA 8.19 muestra un extracto de la configuración del caso de estudio que permite dar soporte a servicios que utilicen Hibernate como ORM.

FIGURA 8.19 – Extracto en la configuración de *beans* para el uso Hibernate

El *bean* dataSource requiere que sean configuradas las propiedades que permiten conectarse a la base de datos relacional.

Estos datos, usualmente, son conocidos únicamente por el DBA, por lo tanto es deseable que sean configurados en un archivo externo a la aplicación.

Ese archivo, para el caso de estudio implementado, se llama **jdbc.properties**, y está ubicado dentro de la carpeta WEB-INF.

La FIGURA 8.20 muestra un extracto de este archivo, que corresponde a los parámetros necesarios para la conexión con la base de datos MySQL del Bug Tracker

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/bugtracker
jdbc.username=root
jdbc.password=

hibernate.dialect=org.hibernate.dialect.MySQLDialect
```

FIGURA 8.20 – Archivo de configuración de los parámetros requeridos por MySQL

La última entrada, **hibernate.dialect**, permite abstraer las interacciones de Hibernate de la base de datos utilizada.

Por ser MySQL la base de datos que se está utilizando, se configura el dialecto de Hibernate *org.hibernate.dialect.MySQLDialect*

E. JAXB y Xstream

Como hemos visto, Java provee los medios para escribir código portable, mientras que XML puede ser usado para definir datos portables.

Como veremos más adelante Apache ServiceMix como implementación de JBI está basado en Java y utiliza XML como el formato de datos que circula dentro del NMR, por lo tanto existe una fuerte relación en Apache ServiceMix entre Java y XML [77]

La FIGURA 8.21 muestra gráficamente como funcionan las herramientas de mapeo XML/Java y Java/XML

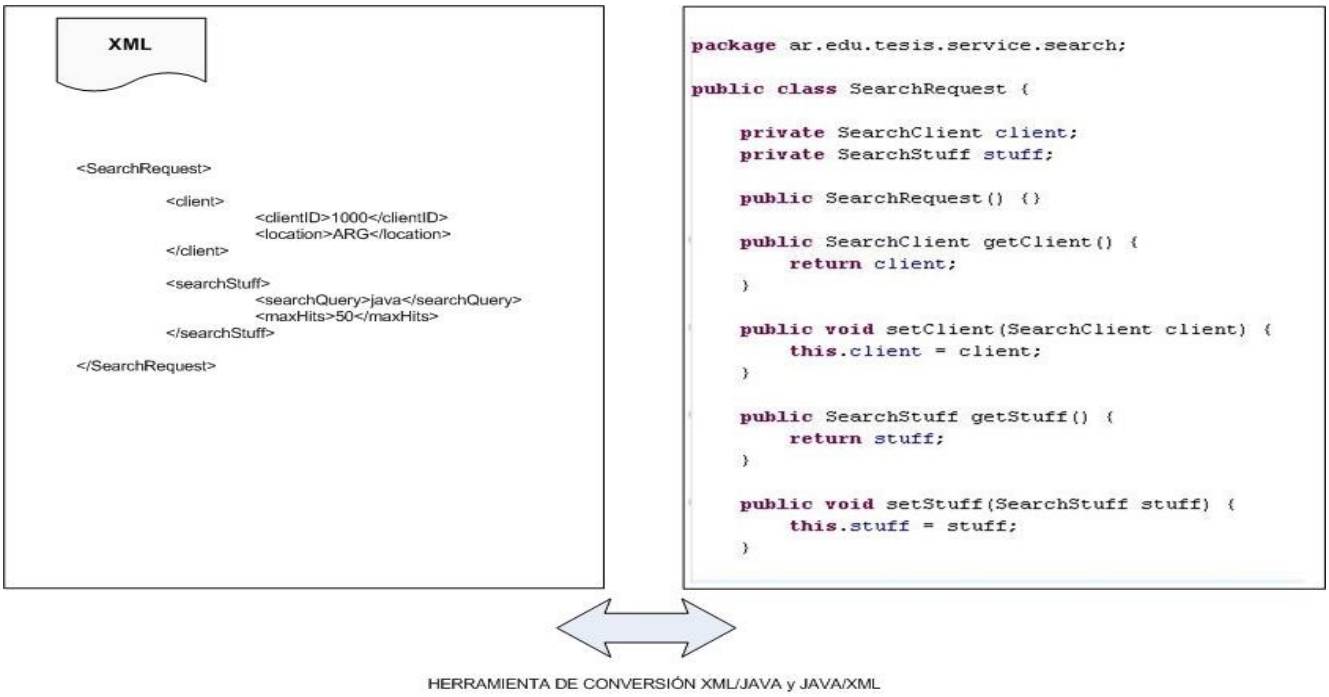


FIGURA 8.21 – Mapeo entre objetos Java y documentos XML

JAXB (Java API for XML Binding): provee una manera ágil de procesar contenido XML usando objetos Java al ligar su esquema XML a una representación en Java. Provee además un API y un conjunto de herramientas que automatizan el mapeo entre documentos XML e instancias Java.

XStream: esta librería también serializa objetos Java en XML y viceversa. Es "liviana" al no necesitar demasiada configuración ni archivos de mapeo. Apache ServiceMix se integra bien con esta librería a través de un API llamado *JavaSource*

F. Apache Log4J

Esta herramienta basada en Java permite realizar el *logging* de las aplicaciones de manera flexible y declarativamente configurable. De esta manera es posible habilitar/deshabilitar la funcionalidad de *logging* en tiempo de ejecución sin modificar el binario de las aplicaciones.

Tiene características distintivas respecto a soluciones similares. En particular se distingue su capacidad de herencia de los *loggers*. Definir una jerarquía de loggers permite tener mayor control sobre que sentencias son registradas en el log, o facilitar distintos niveles de granularidad [70]

La FIGURA 8.22 muestra un extracto del archivo de configuración de log4j utilizado por el caso de estudio implementado.

FIGURA 8.22 – Extracto del archivo de configuración de Apache Log4J

G. JUnit

JUnit es un *framework* que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

JUnit es también un medio de controlar las pruebas de regresión, necesarias cuando una parte del código ha sido modificado y se desea ver que el nuevo código cumple con los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación [80]

H. Apache Lucene

Es una librería que implementa un motor de indexación y búsqueda de textos de alta performance y escalabilidad [68][75]

Sus principales características son:

- Mínimos requerimientos de hardware
- Tamaño del índice es un 20% y 30% menor que el tamaño del texto indexado.
- Múltiples tipos de queries.

- Ordenamiento de resultados
- Búsquedas basadas en rankings
- Búsqueda por campos
- Implementada 100% en Java

Esta librería, y otras que agregan funcionalidades adicionales (Compass Framework[58]), son esenciales en la implementación de los servicios de indexación y búsqueda del caso de estudio.

La FIGURA 8.23 muestra una versión simplificada de la arquitectura de Apache Lucene

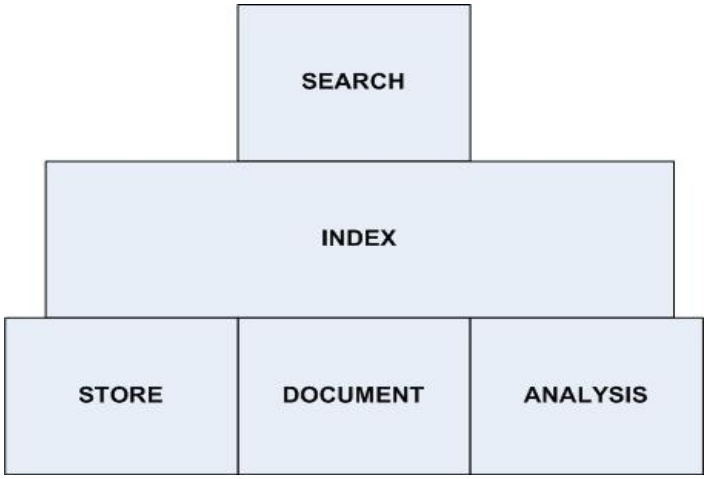


FIGURA 8.23 – Arquitectura de Apache Lucene

En Apache Lucene un índice es, **conceptualmente**, una secuencia de *Documents.*

Un *Document* es una clase específica de Apache Lucene que consiste en un conjunto de *Fields*.

La FIGURA 8.24 muestra gráficamente un índice Apache Lucene, compuesto de M *Documents*, cada uno con N *Fields*

58 **Compass Framework:** http://compass-project.org

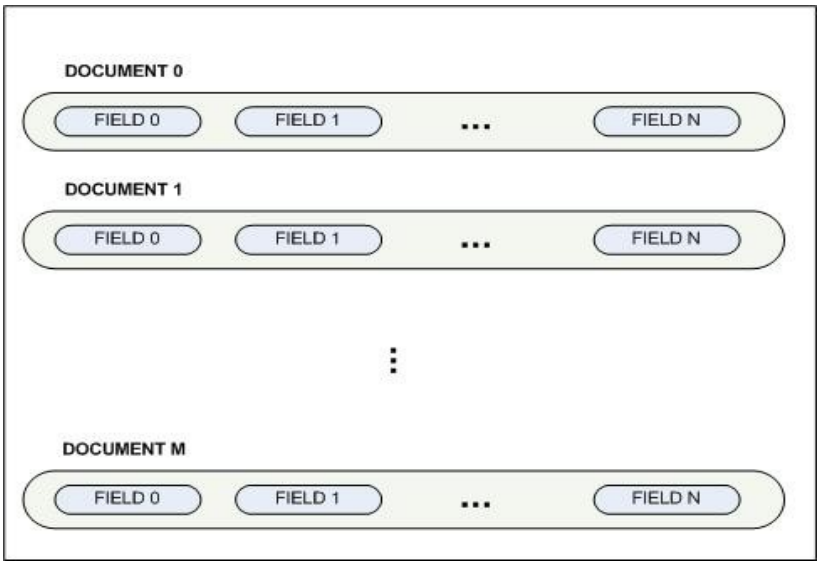


FIGURA 8.24 – Componentes de un índice Apache Lucene

Mientras que un índice, conceptualmente está compuesto de *Documents* y *Fields*, Apache Lucene mantiene el índice físicamente como un conjunto de archivos en un directorio (FIGURA 8.25).

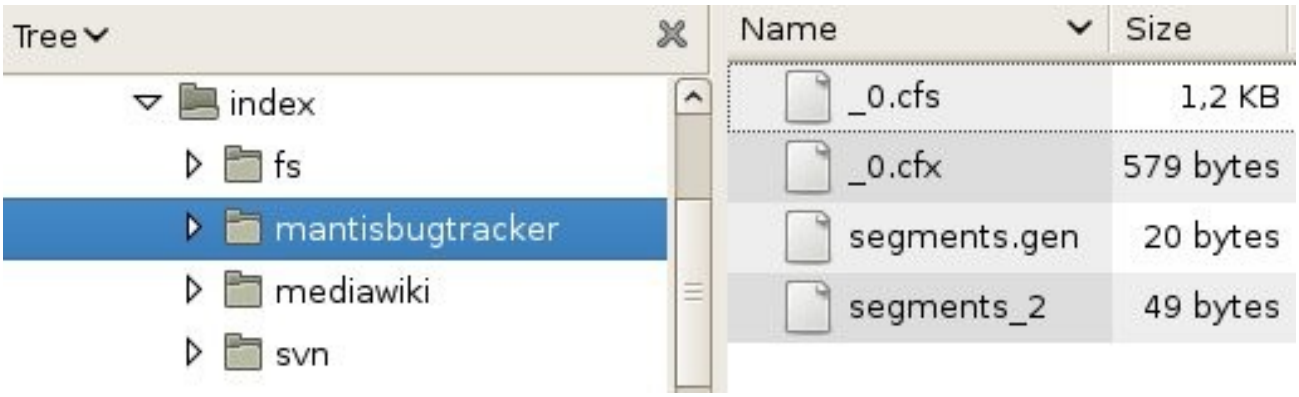


FIGURA 8.25 – Representación del índice Lucene en el file system

Durante la indexación, los *Documents* incorporados al manejador del índice (*IndexWriter)* son mezclados en *segments*; son los *segments* los escritos a archivos en el directorio del índice. Cada *segment* es considerado un subíndice en sí mismo.

La FIGURA 8.26 muestra gráficamente como Apache Lucene almacena físicamente el índice

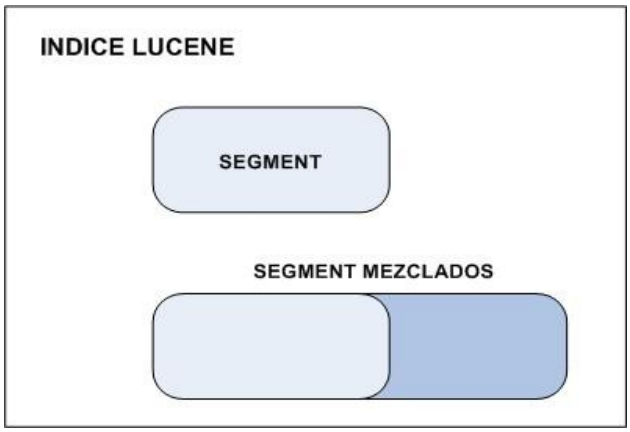


FIGURA 8.26 – Almacenamiento físico del índice Lucene

I. Librería PDFBox

Esta librería es utilizada para trabajar con documentos PDF. Permite la creación de nuevos documentos PDF, la manipulación de documentos existentes y la capacidad de extraer el contenido desde los documentos.

En el caso de estudio implementado es utilizada conjuntamente con Apache Lucene en la extracción del contenido de documentos PDF para su posterior indexación.

J. Librería SVNKit

Esta librería provee un API para la administración general de repositorios subversion.

Entre sus principales características figuran:

- Acceso a través de diferentes protocolos (HTTP/S, SVN, SVN (+ SSH))
- Administración de repositorios locales o remotos: creación, carga, borrado
- Operaciones con las copias locales del repositorio
- Multiplataforma
- No requiere el uso de librerías nativas

En el caso de estudio implementado es utilizada conjuntamente con Apache Lucene para la extracción de contenido indexable del repositorio subversion

K. Apache ActiveMQ[59]

ActiveMQ (FIGURA 8.27) es un sistema de mensajería a nivel empresarial, con soporte para JMS, clustering, persistencia, XA y muchos más.

Fue diseñado para poder ser embebido en otras tecnologías, y se integra fácilmente con Spring y *servlets containers* o contenedores J2EE compatible

ActiveMQ es usado como el sistema de mensajería dentro de la arquitectura del ESB

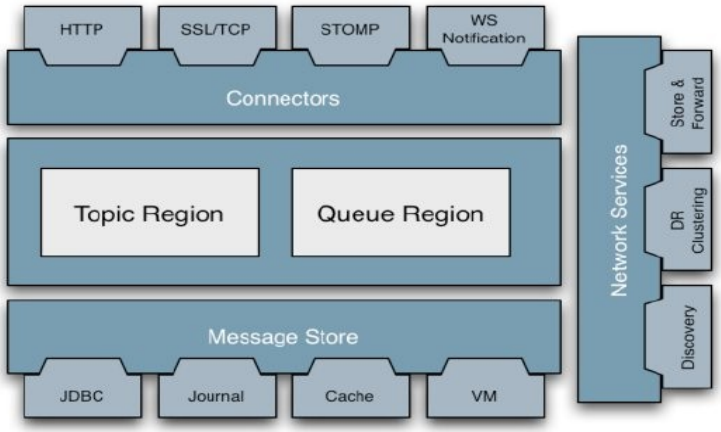


FIGURA 8.27 – Arquitectura de ActiveMQ[60]

L. Apache Camel

Apache Camel[61] permite definir ruteos y reglas de mediación en DSL (*Domain Specific Language*) o a través de archivos de configuración Spring, facilitando la implementación de patrones de integración empresarial (EIP)

En el caso de estudio esta librería se utiliza para la implementación de algunos patrones de integración, fundamentalmente el router basado en contenido (CBR).

M. Apache ServiceMix

M.1. Principales Características

Apache ServiceMix está basado en arquitecturas orientadas a eventos (EDA), y

59 **Apache ActiveMQ**: http://activemq.apache.org

60 Imagen extraída de http://activemq.org

61 **Apache Camel**: http://camel.apache.org

por lo tanto es útil para construir una plataforma para SOI[62]

Es un framework "liviano" que puede ejecutar embebido en otros contenedores, o bien de manera autónoma. Está basado en la especificación JSR 208[54], por lo tanto sus componentes son portables a otros contenedores ESB compatibles con JBI.

Tiene soporte para Spring, por lo tanto es fácil configurar y resolver dependencias entre los componentes y servicios a partir de archivos de configuraciones externos [50][52][62][66]

M.2. Arquitectura

La arquitectura de Apache ServiceMix está basada en el estándar JBI, por lo tanto puede considerarse una "arquitectura abierta" al poder desplegar cualquier componente compatible con JBI.

Este tipo de arquitecturas facilitan la colaboración con otros ESBs, evitando de esta manera los puntos únicos de fallas, y generando contenedores de servicios federados.

M.3. Componentes JBI provistos por Apache ServiceMix

Estos componentes son totalmente compatibles con JBI, soportando el empaquetado y modelo de despliegue de la especificación JSR 208, del JCP.

Los más importantes, la mayoría utilizados en el caso de estudio, son:

- servicemix-bean: permite que los POJOs (*Plain Old Java Object*) participen de intercambios JBI
- servicemix-eip: facilita la implementación de patrones de integración empresariales (EIP).
- servicemix-file: provee integración con el sistema de archivos
- servicemix-http: para comunicaciones HTTP
- servicemix-jms: integración con *middleware* de mensajería a través de JMS
- servicemix-jsr181: facilita el despliegue de POJOs como servicios

M.4. NMR (Normalized Message Router) en Apache ServiceMix

Los componentes conectados al bus, ya sean locales o remotos, interactúan entre si a través del NMR.

---

62 **SOI**: Integración Orientada a Servicios

Apache ServiceMix está basado en principios MOM[63], por lo tanto los mensajes intercambiados utilizan alguno de los MEPs[64] vistos en el capítulo 5 sobre JBI.

M.5. Criterios tenidos en cuenta para la elección de Apache ServiceMix

Entre los criterios tenidos en cuenta en la elección de ServiceMix figuran:

1. Soporte para las funcionalidades básicas de un ESB

- registro de servicios
- conversión de protocolos de transporte
- transformación
- ruteo
- modificabilidad de mensajes
- seguridad
- monitoreo/administración

2. Documentación

- ServiceMix posee documentación de muy buena calidad, principalmente a través de su sitio web, *servicemix.org*
- Se complementa con la documentación de la especificación JBI (JSR 208), muy completa y fácil de leer [54]

3. Comunidad de usuarios activa

- Posee foros[57], y listas de correo[58] con mucha actividad por parte de la comunidad de usuarios
- Tiene un sistema de manejo de bugs[59] que permite conocer la evolución y estado de los errores surgidos, y las ampliaciones sugeridas.

4. Flexibilidad y facilidad para extender con lógica propia

- Al ser una herramienta de código abierto el código fuente[60] está disponible para ser descargado y estudiado.
- La organización de los archivos fuente es natural y entendible (utiliza convención de Apache Maven), lo cual facilita identificar el lugar donde se va a agregar o modificar funcionalidad con código propio.

---

63 **MOM**: Message-oriented Middleware

64 **MEP**: Message Exchange Pattern

5. Soporte de gran cantidad de protocolos de transporte y conectividad

- Es una característica primordial ya que el desarrollo desde cero de conectores para transportes específicos es una tarea que requiere un gran esfuerzo de desarrollo y, principalmente mucha experiencia previa en el área

6. Integración con otros proyectos de código abierto

- Estudiar el código fuente permitió determinar que Apache ServiceMix es fácilmente integrable con los frameworks Spring, Hibernate y Apache Lucene, entre otros.
- La organización del código fuente fue realizada para integrarse al proceso de construcción y administración propuesto por Apache Maven, lo cual brinda sencillez, rapidez y flexibilidad.
- A partir del uso de Apache Maven, la importación de los proyectos en el IDE Eclipse es sencilla.

7. Implementación de JBI

- Apache ServiceMix implementa la especificación JBI (JSR 208) analizada en el capítulo 5

Capítulo 9

jESBihca: Servicios y Componentes

9.1. Servicios y Componentes de jESBihca

El caso de estudio implementado comprende tres partes bien diferenciadas:

1. **Indexación** de datos almacenados en herramientas colaborativas asincrónicas (fuentes de información heterogéneas)

2. **Búsqueda** iniciada por un cliente (humano o PC) que recupera información almacenada en el índice.

3. **Servicios varios** que facilitan la invocación al servicio de búsqueda a partir de diferentes protocolos de transporte y brindan soporte para diferentes escenarios

Como ha sido analizado en capítulos previos, ESB/JBI es una tecnología sencilla, flexible, esencial para la construcción de SOA, que facilita la implementación de servicios reusables e interoperables, permitiendo la creación de aplicaciones al mezclar y relacionar tales servicios.

En la reutilización de código o componentes, se intenta reducir las acciones de "copiar y pegar", promoviendo técnicas OO tales como herencia o composición.

Algo similar ocurre con SOI (*Service-oriented Integration*) donde diferentes servicios son "almacenados" y están disponibles para que sean usados y reutilizados por diferentes aplicaciones, en distintos escenarios, y de diversas formas.

De esta manera, una vez definidos los servicios centrales del dominio en particular, las aplicaciones se construyen componiendo éstos y otros servicios específicos, de manera declarativa.

En el caso de estudio planteado se implementaron diversos servicios compatibles con el estándar JBI, que permiten brindar una solución de integración según el principio de orientación a servicios, y que facilita la implementación de los escenarios que fueron presentados en el capítulo 7, "Caso de Estudio: Análisis".

Los servicios de búsqueda e indexación que implementan el motor de búsqueda full text, junto a servicios que implementan algunos patrones de integración, y otros que brindan soporte a distintos protocolos de transporte y comunicación, constituyen la infraestructura necesaria que permite la búsqueda federalizada en las distintas herramientas colaborativas asincrónicas registradas.

9.2. Búsqueda

A. Introducción

La búsqueda es considerada una noción no muy definida aún, que involucra procesos de computadoras, procesos humanos, pensamientos humanos e incluso sentimientos humanos. Sin embargo, es una funcionalidad que en la actualidad, prácticamente toda aplicación debe contemplar, debido a la cantidad de información a la cual tenemos acceso.

Teniendo en cuenta que la cantidad de información circulante y accesible no detiene su marcha, buscar o, más precisamente, **encontrar** eficientemente la información deseada se ha convertido en un elemento fundamental para las aplicaciones, sistemas e incluso humanos. En IT prácticamente todo gira en torno a la búsqueda y recuperación de información [81]

B. ¿Qué significa Búsqueda?

La interacción de los usuarios o aplicaciones con sistemas de información generalmente sucede por la necesidad de buscar cierta información. Si el usuario o aplicación conoce exactamente que y donde buscar, entonces no seria necesario un servicio o función de búsqueda.

Sin embargo, usualmente este no es el escenario típico. En particular, algunos usuarios no saben exactamente **qué** están buscando; tienen ideas difusas, generalmente desorganizadas, y realizan la búsqueda esperando basada en esas ideas.

Si los resultados devueltos son demasiado grandes, es posible que el objetivo de

la búsqueda se "pierda" entre tanta información; por el contrario, si la cantidad de respuestas es pequeña, es posible que la información buscada haya sido filtrada.

Los servicios de búsqueda típicamente deben enfrentarse a búsquedas que son realizadas con diferentes grados de "claridad" por parte de los usuarios y aplicaciones. Búsquedas realizadas con mayor exactitud significan resultados retornados según su **relevancia**[65]

Sin embargo, aunque los usuarios o las aplicaciones conozcan exactamente **qué** información están buscando, es posible que no sepan **dónde** buscar y **cómo** acceder a dicha información.

Existen diferentes estrategias para implementar la funcionalidad de búsqueda

1. **Categorización de la información:** esta estrategia es la más sencilla y consiste en categorizar la información en estructuras tipo árbol. En este caso se intenta anticipar los tipos de búsquedas realizados. Esta estrategia está pensada para que el servicio de búsqueda sea accedido por humanos.

2. **Utilizar una pantalla de búsqueda detallada**: otra estrategia enfocada en el usuario humano, es presentar una pantalla con varios criterios que permiten aplicar filtros a los datos almacenados. Este sistema es útil cuando el usuario sabe exactamente qué está buscando, no así para aquellos que no tienen tal "claridad".

3. **Utilizando un campo de texto simple:** la tercer estrategia es la más simple desde el punto de vista del cliente del servicio de búsqueda, y consiste un campo de texto simple que oculta la complejidad de los datos y modelo de datos subyacente. Permite al usuario humano, o aplicación, expresar los términos de la búsqueda libremente. Sin embargo, la simplicidad del *front-end* implica mayor complejidad del *back-end*. En este caso se invierten los roles; ya no es el usuario quien usa el "lenguaje" del sistema, sino que el sistema debe comprender el lenguaje del usuario.

4. La cuarta estrategia posible es una solución donde se **mezclan las estrategias anteriores.**

9.3. Estrategias de implementación de un Motor de Búsqueda

A. Motor de búsqueda basado en SQL

La elección de la estrategia utilizada es el primer paso. El siguiente es lograr implementar esa estrategia, de manera eficiente, escalable y flexible.

65 **Relevancia**: significa retornar la información que se considera más cercana a la respuesta buscada en los primeros lugares de la lista.

Típicamente, en aplicaciones que utilicen base de datos relacionales como repositorio de los datos, el servicio de búsqueda es implementado a partir de consultas SQL a la base de datos, construidas según el texto de búsqueda provisto por el usuario.

SQL es una poderosa y consolidada herramienta para la recuperación de información. Sin embargo no es la herramienta adecuada para búsquedas realizadas a partir de diferentes tipos de textos ingresados por usuarios o enviados por aplicaciones.

Suponiendo que el texto ingresado manualmente por un usuario, o recibido desde una aplicación cliente, es *textoDeBusqueda*, y la información esta almacenada en una base de datos relacional con M tablas, de N campos cada una, el motor de búsqueda basado en SQL debe buscar los posibles resultados en las tablas y columnas existentes, eventualmente uniéndolas en complejas consultas.

Por ejemplo una posible consulta pseudo SQL para este escenario tendría la forma de un *SELECT campoID, campoX FROM*, el operador *like* junto a caracteres *wild card ("%")* y el texto de búsqueda ingresado

```
Select campoID, campoX
From Tabla1 join Tabla2 join ... join TablaM
Where campo1 like textoDeBusqueda% or campo2 like
textoDeBusqueda% or  ....  or campoN like textoDeBusqueda%
```

Ejemplo E.1

Esta estrategia quizás funcione para textos de búsqueda simples, que puedan ser comparados con los distintos campos de las diferentes tablas. Sin embargo si el texto de búsqueda es una frase, incluso con errores ortográficos, esta técnica quizás no sea la adecuada. Posiblemente sea necesario dividir la frase en palabras sueltas, y buscar en las diferentes tablas y columnas por separado.

A partir de la separación en palabras de la frase original, surge la necesidad de filtrar el "ruido" generado por esta tarea.

Por ejemplo, si la frase de búsqueda ingresada hubiese sido "*java es un lenguaje de programación orientado a objetos*", la división en palabras generaría la lista de palabras "java", "es", "un", "lenguaje", "de", "programación", "orientado", "a", "objetos".

En este caso particular, el "ruido" producido por la separación en palabras de la frase son las palabras que sintácticamente corresponde a preposiciones y artículos ("*de*", "*a*", "*un*"). Si se incluyen estas palabras en la búsqueda se van a incluir resultados no relacionados con el espíritu original de la búsqueda

Luego, por lo visto hasta el momento la implementación de un motor de búsqueda basado en SQL tiene 2 limitaciones. Por un lado el uso de textos de búsquedas que sean frases dificulta la recuperación de resultados que pudiesen ser útiles, posiblemente por faltas de ortografía o frases sintácticamente incorrectas.

La solución a esta limitación puede ser dividir la frase en palabras sueltas. Según

lo analizado, esta técnica necesita de una solución que permita filtrar palabras comunes que no ayuden a encontrar los resultados deseados.

Una vez restringidas las palabras a una lista significante para la búsqueda realizada el motor de búsqueda SQL puede buscar cada palabra en las diferentes columnas.

Como vimos en el ejemplo anterior, la búsqueda de palabras dentro del contenido de una columna requiere un operador SQL no primitivo (*like*), utilizado con caracteres *wildcard* ("*%*" en particular). Esta operación puede resultar muy costosa si requiere una recorrida total de las tablas, o puede optimizarse mediante el uso de índices.

El índice es una estructura de datos que permite hacer más eficiente las búsqueda al ordenar la estructura de datos de acuerdo al valor de la columna.

Para el ejemplo E.1 la base de datos puede beneficiarse con el uso de indices, y ejecutar con bastante eficiencia la consulta, al estar los datos ordenados alfabéticamente en la estructura de datos del índice.

Sin embargo, para que la consulta sea más genérica es necesario utilizar el carácter *wildcard* en ambos extremos del texto a buscar.
El ejemplo E.1 debe reescribirse de la siguiente manera:

```
Select campoID, campoX From Tabla1 join Tabla2 join ... join TablaM
Where campo1 like %textoDeBusqueda% or campo2 like
%textoDeBusqueda% or .... or campoN like %textoDeBusqueda%
```

Ejemplo E.2

En este caso la base de datos no puede beneficiarse en su totalidad de la existencia de indices asociados a las columnas, ya que el valor de la columna puede no empezar con el texto de búsqueda *textoDeBusqueda.*

En ocasiones, la base de datos utiliza igualmente el indice para realizar un procedimiento similar realizado en las tablas, en caso de la no existencia de indices. En esta situación realiza una recorrida por todo el indice buscando las claves que coincidan en alguna parte del contenido con el texto a buscar. Aunque es necesaria una recorrida secuencial de todas las claves, esta técnica es un poco más eficiente que recorrer las filas de las tablas ya que el indice es una estructura más compacta.

Búsqueda de palabras con significado similar

Hasta ahora el motor de búsqueda SQL necesita implementar consultas SQL complejas, de baja performance, con cierto preprocesamiento Java para poder realizar búsqueda de frases, separadas en palabras, contenidas en el texto de determinadas columnas y tablas.

Sin embargo, aún existen deficiencias para lograr un motor de búsqueda óptimo.

Las palabras dentro de la frase provista por el cliente (usuario o aplicación) pueden contener errores tipográficos, o no coincidir letra por letra con la palabra almacenada en la base de datos. Es decir, el cliente espera que el motor de búsqueda retorne resultados que contengan las palabras deseadas, pero también variaciones de dichas palabras.

Volviendo al ejemplo donde la frase de búsqueda era "*java es un lenguaje de programación orientado a objetos*", y luego de separar esta frase en palabras individuales, es posible que el cliente también espere obtener entre los resultados devueltos aquellas filas donde sus columnas tengan en su contenido palabras "similares" a las originales. Por ejemplo, puede ser deseable que entre los resultados aparezcan las filas que tengan la palabra "orientación" en determinadas columnas en vez de la palabra original "*orientado*"

Esta técnica de identificar la raíz de las palabras se conoce como *stemming process.* Más adelante en esta sección se analizará la herramienta utilizada para implementar este proceso de recuperación de palabras derivadas de las originales

Este proceso de *stemming* puede ser ampliado para incluir no sólo derivaciones de palabras, sino también sinónimos.

Estos nuevos requerimientos profundizan las limitaciones de los motores de búsqueda SQL, lo cual refuerza la necesidad de buscar herramientas alternativas de implementación.

Búsqueda de palabras con errores tipográficos

En la frase de búsqueda de ejemplo, es posible que el cliente (usuario o aplicación) ingrese o genere la frase de búsqueda con errores tipográficos. Por ejemplo, si la frase enviada es: "java es un lengua**g**e de programación orientado a objetos", donde la palabra "*lenguaje*" tiene un error tipográfico, es deseable que el motor de búsqueda pueda recuperar ese error y retornar los resultados esperados. El motor de búsqueda SQL no ofrece una solución simple a este problema.

Relevancia

Un último aspecto, quizás el más importante, es la relevancia en los datos devueltos. Suponiendo que el motor de búsqueda retorne los resultados esperados, es posible que la cantidad devuelta sea demasiado grande, siendo "incómodo" rastrear la información buscada.

Ordenar los resultados por una propiedad en particular tampoco soluciona la problemática de encontrar entre todos los resultados devueltos la información buscada. El motor de búsqueda debe poder ordenar los resultados en base a su **relevancia.**

Existen algunas estrategias para implementar este aspecto, de acuerdo a

diferentes semánticas dadas al término **relevancia.**

Una estrategia puede considerar que un dato es **más relevante que** otro si se encuentra en determinada columna y no en otras. Por ejemplo, si en una fila la palabra "*java*" se encuentra en la columna "*lenguaje*", mientras que en otra se encuentra en la columna "*descripción*", según esta técnica, la primer fila podría tener mayor relevancia, y aparecería primero en la lista de resultados retornados.

Otra estrategia puede basarse en la cantidad de apariciones de la palabra en determinadas columnas. Mientras mayor es el número de apariciones, mayor es su relevancia.

Es posible considerar de mayor relevancia a las columnas que contengan la palabra exacta, respecto a otras que contengan aproximaciones (derivaciones o sinónimos) de la palabra original.

También se considera de mayor relevancia las filas que en sus columnas contienen la mayor cantidad de palabras de la frase de búsqueda original, e incluso la cercanía entre éstas.

Los motores de búsqueda SQL no ofrecen una solución simple a estos requerimientos: proximidad de palabras, número de palabras concordantes, entre otros.

B. Motor de Búsqueda *Full Text*

Esta tecnología se basa en encontrar documentos que contengan un conjunto de palabras. De esta manera permite solucionar las limitaciones encontradas en los motores de búsqueda basados en SQL [82]

El objetivo principal de estos motores de búsqueda es, dado un conjunto de palabras (*query*), debe retornar los documentos que contengan esas palabras, o un subconjunto de las mismas.

La búsqueda secuencial de esos documentos en todas las fuentes de información registradas es un proceso ineficiente, por lo tanto el motor de búsqueda *full text* divide su funcionalidad en dos operaciones principales: **indexar** la información en un formato de almacenamiento eficiente y **búsqueda** de la información relevante en el índice construído previamente. La noción de **palabra** es esencial en estos motores de búsqueda, siendo considerada la pieza de información atómica manipulada por el motor.

1. Indexación

Indexar comprende múltiples operaciones cuyo objetivo es construir una estructura que facilite y haga más eficiente la búsqueda de información.

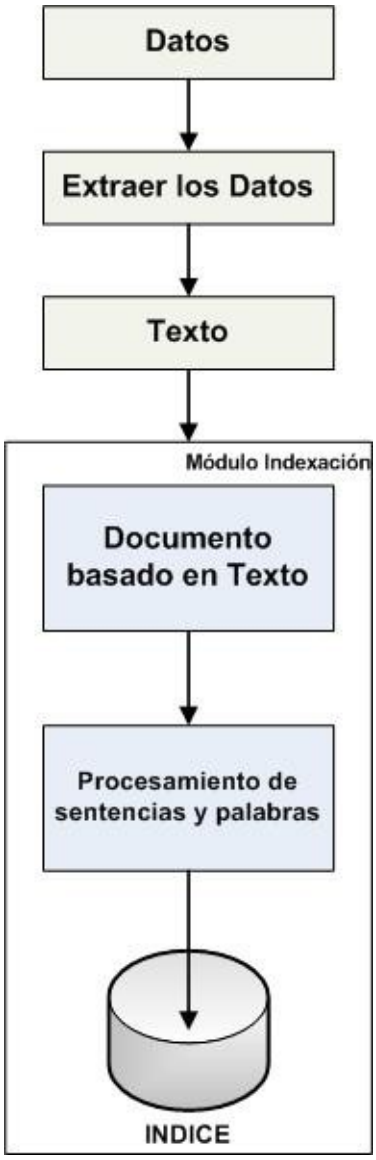


FIGURA 9.1 – Componentes involucrados en el proceso de indexación

La FIGURA 9.1 muestra gráficamente los componentes típicos de los motores de búsqueda full text, involucrados para la indexación.

La primer operación necesaria es obtener la información que quiere indexarse. Esta información puede ser extraída de base de datos, de sitios web, de documentos con diferentes formatos (PDF, ODF, DOC, XML, otros). Una vez extraída la información, ésta es procesada.

El segundo paso convierte la información original en una representación especial del texto: *document.* Un *document* es un contenedor que mantiene la representación en texto de los datos originales, ya sea que representan un archivo HTML, XML, fila de una tabla u otros. Sin embargo, no todos los datos necesitan ser incorporados al *document,* sino la información útil para realizar búsquedas posteriores.

Opcionalmente este proceso puede categorizar los datos, almacenándolos en diferentes propiedades (*fields).* Puede pensarse un *document* como un conjunto de *fields.*

La tercer operación procesará el texto de cada propiedad (*field*) y extraerá la pieza de información atómica que el motor de búsqueda *full text* entiende: **palabras**.

Esta operación es clave para lograr una buena performance del motor de búsqueda, y para dar solución a algunas limitaciones encontradas en los motores de búsqueda SQL: búsquedas de derivaciones (*stemming*) y búsquedas de sinónimos.

El último paso del proceso de indexación es el almacenamiento de los *documents* (opcional), y la creación de la estructura optimizada que hará más eficientes las búsquedas posteriores.

La estructura del índice almacena las palabras individuales e información adicional por cada palabra, útiles para conocer la relevancia.

Entre esta información adicional se encuentran la frecuencia de la palabra, su posición y corrimiento (*offset*).

Con esta información adicional el motor de búsqueda *full text* puede determinar que tan "relevante" es una palabra en un documento.

2. Búsqueda

La búsqueda toma como entrada una consulta (*query*) desde un cliente (usuario final o aplicación), y devuelve la lista de resultados que concuerden con la búsqueda realizada, preferentemente ordenados según su relevancia.

Al igual que el proceso de indexación la búsqueda implica múltiples pasos, que eliminan las limitaciones vistas anteriormente en el motor de búsqueda SQL.

La FIGURA 9.2 muestra gráficamente los pasos involucrados en el proceso de búsqueda

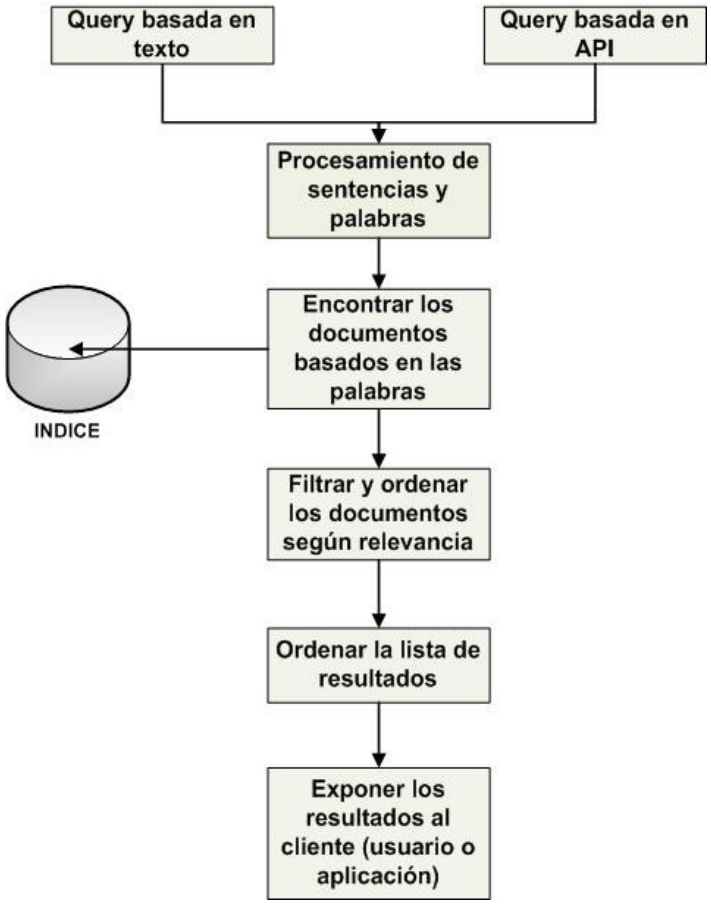


FIGURA 9.2 – Pasos involucrados en el proceso de búsqueda

La primer operación realizada es la obtención de la consulta por parte del cliente, ya sea el usuario final a través de un navegador web, por ejemplo, o bien desde una aplicación a partir de un API que exponga la funcionalidad del motor de búsqueda

La segunda operación es responsable de tomar las sentencias/listas de palabras y aplicar la operación similar a la realizada por el proceso de indexación (división en palabras, *stemming*). Este paso es critico ya que es el lenguaje en común entre las operaciones de indexación y búsqueda que permite que ambos procesos sea uniformes. Si el mismo conjunto de operaciones no es aplicado a ambos procedimientos, entonces la búsqueda no encontrara las palabras indexadas.

En base a este lenguaje en común entre la indexación y búsqueda, la tercera operación leerá el indice y recuperara la información asociada a cada palabra que concuerde con la consulta realizada. Como fue explicado anteriormente, el indice almacena por cada palabra la lista de documentos que la contienen, la frecuencia y posición de la palabra en el documento.

La tercera operación se realiza de manera **eficiente** ya que el documento **no** es cargado para determinar si contiene palabras que concuerden con la consulta.

La cuarta operación procesara la información recuperada del indice y construida

la lista de manejadores de los documentos. De la información que existe sobre las palabras (cantidad de documentos que contienen determinada palabra, frecuencia y posición de palabras) el motor de búsqueda filtra algunos documentos y calcula un resultado (*score*) para cada documento. Mientras mayor sea este valor, más alto figurará en la lista de resultados retornados.

Una vez obtenida la lista ordenada de resultados, el motor de búsqueda expone los resultados al cliente, ya sea a través de una interface web (para usuarios finales), o a través de un API.

B.1. Estrategias de implementación de un motor de búsqueda Full Text

Básicamente existen tres estrategias de implementación de un motor de búsqueda full text

B.1.1. Motor de búsqueda full text embebido en una base de datos relacional

El objetivo de esta estrategia es ampliar las consultas SQL de las aplicaciones con características propias de los motores full text.

Las principales ventajas de esta estrategia son:

- Los datos e indice son administrados por el mismo producto
- Los datos e indices están siempre sincronizados
- La flexibilidad de SQL y de las búsquedas full text brindan gran potencialidad a esta aproximación

La FIGURA 9.3 muestra gráficamente la implementación de esta estrategia

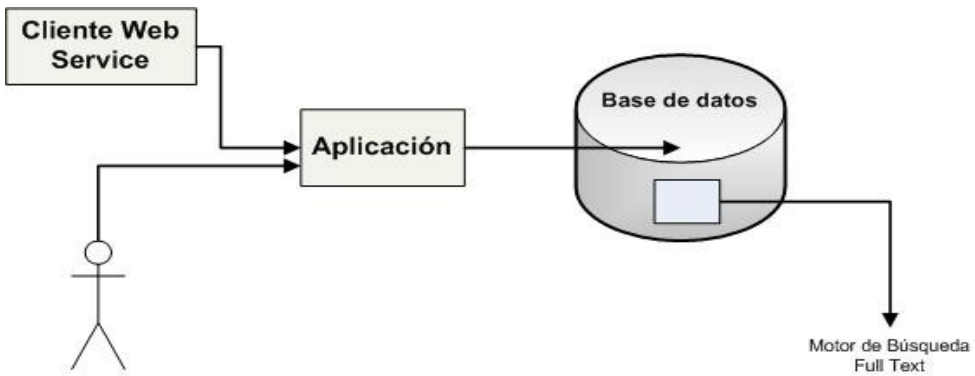


FIGURA 9.3 – Motor de búsqueda full text embebido en una base de datos

Entre los principales inconvenientes de esta solución se encuentran el consumo de recursos de la base de datos por el motor de búsqueda full text, que es un proceso intensivo en cuanto a ciclos de CPU y operaciones de entrada/salida.

Y también su falta de portabilidad, al no existir por el momento estándares para búsquedas full text.

B.1.2. Motor de búsqueda full text “como caja negra”

Otra alternativa son los productos cerrados dedicados a la indexación y búsqueda de contenido en ambientes heterogéneos, tales como sitios web, intranet y sistemas de información en general.

Estas soluciones facilitan la búsqueda de datos, principalmente a través de las áreas de una organización.

En las alternativas comerciales de estos productos se establece el pago de licencia en base a la cantidad de documentos indexados, o el tamaño del índice.

Aunque potentes y escalables, estas soluciones en general carecen de la flexibilidad para ser embebidas en aplicaciones que requieran brindar el servicio de búsqueda full text.

B.1.3. Motor de búsqueda full text como librerías

La tercer opción, y la adoptada por el caso de estudio implementado en este trabajo, es el uso de librerías que ofrecen la funcionalidad de búsqueda full text, y son fácilmente embebidas en las aplicaciones que necesiten integrar dicha funcionalidad.

De esta manera la aplicación realizará invocaciones al API de la librería para las operaciones de indexación y búsqueda

Esta alternativa ofrece gran flexibilidad en las posibilidades de expresar una consulta, las características a exponer del motor de búsqueda, control sobre el proceso de indexación y su estructura y filtros para evitar que ciertos datos sean buscados, entre otras funcionalidades adicionales.

La FIGURA 9.4 muestra gráficamente el uso de esta alternativa

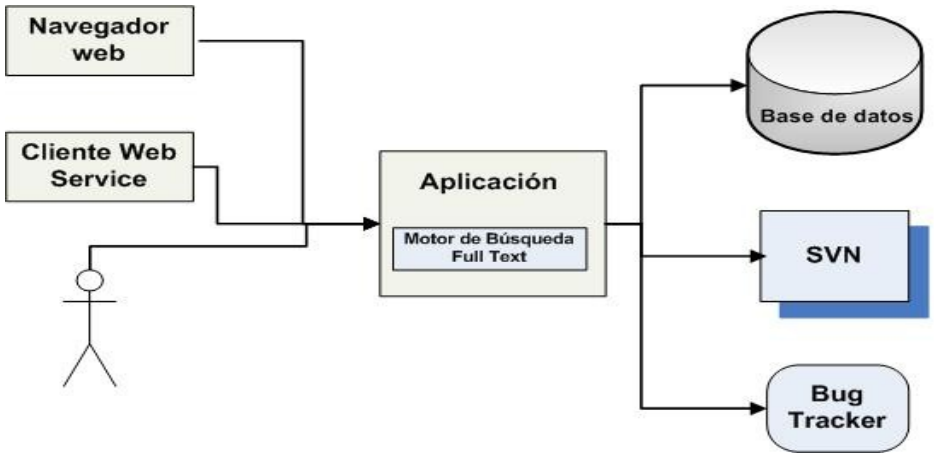


FIGURA 9.4 – Librería que ofrece la funcionalidad de búsqueda full text

Entre las librerías existentes, Apache Lucene es probablemente la más popular, y la elegida para la implementación del motor de búsqueda del caso de estudio.

9.4. Implementación del Motor de Búsqueda del Caso de Estudio

A. Servicio de Indexación

El proceso de indexación es implementado como un servicio separado, el cual es invocado periódicamente (a través del Servicio de Scheduling; ver sección 9.10) lo cual permite mantener el índice actualizado**.**

La FIGURA 9.5 muestra un extracto de código correspondiente al servicio de indexación

```
public synchronized void doIndex() {

    long beginTime = System.currentTimeMillis();

    log.info("INICIANDO PROCESO DE INDEXACION");

    if (!adapters.isPerformingIndexOperation()) {
            adapters.index();
    }else {
        log.info("..... INDEXACION AUN EN PROCESO ..... ");
    }

    long indexTime = System.currentTimeMillis() - beginTime;

    log.info("FINALIZANDO PROCESO DE INDEXACION EN: " + indexTime);
```

FIGURA 9.5 – Extracto de código correspondiente al servicio de indexación del caso de estudio

Este servicio indexa las fuentes de información registradas, a través del uso de adaptadores específicos El caso de estudio planteado contempla 4 herramientas colaborativas a indexarse: MediaWIKI, Mantis BT, Subversion y File System**.**

Para cada una de estas herramientas es implementado un adaptador específico que permite al servicio indexador recuperar la información relevante que el caso de estudio requiere.

A.1. Adaptadores a las fuentes de información registradas

A.1.1. Adaptador a MediaWIKI

El adaptador a MediaWIKI se implementa a partir de una conexión JDBC (FIGURA 9.6) a la base de datos de soporte a esta herramienta; en este caso la información generada por MediaWIKI se almacena en una base de datos MySQL.

```
<bean id="mediaWikiDS" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>${mediaWiki.driverClassName}</value></property>
    <property name="url"><value>${mediaWiki.url}</value></property>
    <property name="username"><value>${mediaWiki.username}</value></property>
    <property name="password"><value>${mediaWiki.password}</value></property>
</bean>
```

FIGURA 9.6 – Configuración parámetros de la base de datos MySQL de MediaWIKI

Luego el servicio de indexación a través de archivos de configuración XML, especifica, de manera declarativa, la consulta SQL para la recuperación de la información a ser indexada. La FIGURA 9.7 muestra la declaración de la consulta SQL utilizada por este adaptador

```
<bean id="mediaWikiMapping" class="org.compass.gps.device.jdbc.mapping.ResultSetToResourceMapping">

    <property name="alias"><value>mediaWiki</value></property>
    <property name="selectQuery">
    <value>
      SELECT CONCAT('${mediaWiki.url_prod}index.php/',page_title) as mw_url,
             page_id, si_text, SUBSTRING(si_text, 1, 50) as mw_summary, page_title as mw_title
      FROM page, searchindex
      WHERE page_id=si_page  AND page_is_redirect=0 AND page_namespace IN (0)
    </value>
    </property>
```

FIGURA 9.7 – Consulta SQL para recuperar los datos de MediaWIKI a indexar

Los datos devueltos por la consulta SQL luego son transformados a un formato de documento (FIGURA 9.8) entendible por Apache Lucene, con todas las propiedades (autor, datasource, URL, summary, entre otras) requeridas por los 3 escenarios implementados en el caso de estudio.

```
<property name="indexUnMappedColumns"><value>false</value></property>
 <property name="idMappings">
  <list>
    <bean class="org.compass.gps.device.jdbc.mapping.IdColumnToPropertyMapping">
      <property name="columnName"><value>page_id</value></property>
      <property name="propertyName"><value>ID</value></property>
     </bean>
  </list>
  </property>
 <property name="dataMappings">
  <list>
    <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
      <property name="columnName"><value>si_text</value></property>
      <property name="propertyName"><value>contents</value></property>
    </bean>
    <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
      <property name="columnName"><value>mw_summary</value></property>
      <property name="propertyName"><value>summary</value></property>
    </bean>
     <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
      <property name="columnName"><value>mw_url</value></property>
      <property name="propertyName"><value>url</value></property>
    </bean>
    <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
      <property name="columnName"><value>mw_title</value></property>
      <property name="propertyName"><value>title</value></property>
    </bean>
  </list>
</property>
```

FIGURA 9.8 – Configuración de los campos de los documentos entendibles por Apache Lucene

A.1.2. Adaptador a Mantis BT

El adaptador a Mantis BT es implementado de manera similar al adaptador a MediaWIKI, ya que también utiliza una base de datos MySQL para almacenar la información generada por la aplicación La configuración de conexión a la base de datos se realiza a partir de un archivo de configuración externo (FIGURA 9.9)

```
<bean id="mantisDS" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"><value>${mantis.driverClassName}</value></property>
    <property name="url"><value>${mantis.url}</value></property>
    <property name="username"><value>${mantis.username}</value></property>
    <property name="password"><value>${mantis.password}</value></property>
</bean>
```

FIGURA 9.9 - Configuración parámetros de la base de datos MySQL de Mantis BT

El servicio de indexación a través de un archivo de configuración XML declara la consulta SQL (FIGURA 9.10) utilizada para recuperar la información relevante a ser indexada.

```
<bean id="mantisMapping" class="org.compass.gps.device.jdbc.mapping.ResultSetToResourceMapping">

      <property name="alias"><value>mantisBugTracker</value></property>
      <property name="selectQuery">
      <value>

      SELECT
            SUBSTRING(bt.summary, 1, 50) as bt_summary,

            bt.id as bug_id,
            CONCAT(bt.summary, ' ', btt.description,' ', COALESCE(btt.additional_information, ' '), ' ', COALESCE(bntt.note, ' '))
                                                                                                        as bt_contents,

            COALESCE(SUBSTRING(btt.description, 1, 20), CONCAT(bt.id, ' - ', SUBSTRING(bt.summary, 1, 20))) as bt_project_description,

            CONCAT('${mantis.url_prod}view.php?id=',bt.id) as bt_url


        FROM ( ( ( (mantis_bug_table as bt left join mantis_project_table as pt on bt.project_id=pt.id)
             left join  mantis_bug_text_table as btt on btt.id=bt.bug_text_id )
             left join mantis_bugnote_table as bnt on bnt.bug_id=bt.id)
             left join mantis_bugnote_text_table  as bntt on bntt.id=bnt.bugnote_text_id)

      </value>
      </property>
```

FIGURA 9.10 - Consulta SQL para recuperar los datos de Mantis BT a indexar

Lo mismo que sucede con MediaWIKI, los datos devueltos por la consulta SQL deben ser transformados a un formato entendible por Apache Lucene. La FIGURA 9.11 muestra el extracto de configuración correspondiente al adaptador de Mantis BT

```
<property name="idMappings">
 <list>
   <bean class="org.compass.gps.device.jdbc.mapping.IdColumnToPropertyMapping">
     <property name="columnName"><value>bug_id</value></property>
     <property name="propertyName"><value>ID</value></property>
   </bean>
 </list>
 </property>
<property name="dataMappings">
 <list>
   <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
     <property name="columnName"><value>bt_contents</value></property>
     <property name="propertyName"><value>contents</value></property>
   </bean>
   <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
     <property name="columnName"><value>bt_summary</value></property>
     <property name="propertyName"><value>summary</value></property>
   </bean>

   <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
     <property name="columnName"><value>bt_url</value></property>
     <property name="propertyName"><value>url</value></property>
   </bean>

   <bean class="org.compass.gps.device.jdbc.mapping.DataColumnToPropertyMapping">
     <property name="columnName"><value>bt_project_description</value></property>
     <property name="propertyName"><value>title</value></property>
   </bean>
```

FIGURA 9.11 - Configuración de los campos de los documentos entendibles por Apache Lucene

A.1.3. Adaptador a Subversion

El adaptador al repositorio Subversion es implementado utilizando la librería SVNKit. La URL al repositorio a ser indexado es configurada externamente e "inyectada" en el adaptador a través del framework Spring (FIGURA 9.12) .

```
<bean abstract="true" id="svnAdapter" class="ar.edu.tesis.adapters.impl.SVNAdapterImpl">
    <property name="textExtractor" ref="esb-text-extractor" />
</bean>

<bean id="svnGpsDevice" parent="svnAdapter">
        <property name="name"><value>svnAdapter</value></property>
        <property name="projectName"><value></value></property>
        <property name="repoURL"><value>file:///usr/local/apache-servicemix-3.2.2/svnrepo/</value></property>
        <property name="compass"><ref local="localCompass" /></property>
</bean>
```

FIGURA 9.12 - Definición de beans que configuran el adaptador a Subversion.

Luego se realiza una iteración entre todos los elementos contenidos en el repositorio, y se transforman a documentos (FIGURA 9.13) con formato entendible por Apache Lucene, con las propiedades requeridas por los 3 escenarios implementados en el caso de estudio planteado.

```
String url = this.getRepoURL() + File.pathSeparator + fileURL;

r.addProperty("uid", fileURL);
r.addProperty("creator", entry.getAuthor() );
r.addProperty("datasource", this.getName());
r.addProperty("url", url);
r.addProperty("modified", entry.getDate());
r.addProperty("creationdate", creationDate);
r.addProperty("type", extension);
r.addProperty("keywords", keywords);
r.addProperty("title", fileURL);

Object[] res = SvnUtils.getFile(repo, projectName + File.separator + entry.getRelativePath());

if (res != null) {

Document doc = this.getTextExtractor().doGetDocument(res);
String contenido = doc.getField("contents").stringValue();
if (contenido == null)
    contenido = "";

SVNRepository repository = SvnUtils.getRepositoryForFile(repoURL +  entry.getName(), "", "");
String logMessage = SvnUtils.getAllLogs(repository, null, null);

logMessage = (logMessage == null?"":logMessage);

contenido += " " + logMessage;

int summarySize = Math.min( (contenido == null?0: contenido.length()), 50 );
String summary = (contenido == null?"": contenido.substring(0, summarySize));

r.addProperty("summary", summary);

if (contenido != null)
    r.addProperty("contents", contenido);
```

FIGURA 9.13 – Transformación de las propiedades de los documentos del repositorio a documentos entendibles por Apache Lucene

En un archivo XML externo (FIGURA 9.14) se realiza la configuración de los diferentes campos que conforman el documento Lucene a ser indexado

```
<resource alias="SVN">

      <resource-id name="uid" />
      <resource-property name="creator" store="yes" index="un_tokenized"/>
      <resource-property name="contents" store="no" index="tokenized"/>
      <resource-property name="summary" store="yes" index="no"/>
      <resource-property name="url" store="yes" index="un_tokenized"/>
      <resource-property name="type" store="yes" index="un_tokenized"/>
      <resource-property name="datasource" store="yes" index="un_tokenized"/>
      <resource-property name="modified" store="yes" index="un_tokenized"/>
      <resource-property name="creationdate" store="yes" index="un_tokenized"/>
      <resource-property name="keywords" store="yes" index="un_tokenized"/>
      <resource-property name="title" store="yes" index="un_tokenized"/>

</resource>
```

FIGURA 9.14 – Configuración de los campos a ser indexados por Apache Lucene

A.1.4. Adaptador del Sistema de Archivos

El adaptador al sistema de archivos es implementado a partir de analizadores a distintos tipos de archivos (pdf, txt, xls, doc, ppt, html, xml, entre otros) que se encargan de recuperar el texto que será indexado para su posterior búsqueda

Estos analizadores son configurados en un archivo XML externo (FIGURA 9.15) de manera declarativa

```
<beans  xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="
           http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">



 <bean id="documentHandlerManager" class="ar.edu.tesis.indexing.documents.handler.DefaultDocumentHandlerManager">

     <property name="extensionHandler">
        <map>
            <entry key="pdf"><value>ar.edu.tesis.indexing.documents.handler.file.pdf.PdfBoxDocumentHandler</value></entry>
            <entry key="doc"><value>ar.edu.tesis.indexing.documents.handler.file.msoffice.MsWordDocumentHandler</value></entry>
            <entry key="xls"><value>ar.edu.tesis.indexing.documents.handler.file.msoffice.MsExcelDocumentHandler</value></entry>
            <entry key="ppt"><value>ar.edu.tesis.indexing.documents.handler.file.msoffice.MsPowerPointDocumentHandler</value></entry>
            <entry key="rtf"><value>ar.edu.tesis.indexing.documents.handler.file.rtf.RtfDocumentHandler</value></entry>
            <entry key="txt"><value>ar.edu.tesis.indexing.documents.handler.file.text.TextDocumentHandler</value></entry>
            <entry key="java"><value>ar.edu.tesis.indexing.documents.handler.file.text.TextDocumentHandler</value></entry>
            <entry key="xml"><value>ar.edu.tesis.indexing.documents.handler.file.text.XMLDocumentHandler</value></entry>
            <entry key="html"><value>ar.edu.tesis.indexing.documents.handler.file.html.HTMLDocumentHandler</value></entry>

        </map>
    </property>

</bean>
```

FIGURA 9.15 – Configuración de analizadores de documentos para distintos formatos de archivos

La información recuperada de los distintos archivos es luego transformada a un documento con un formato entendible (FIGURA 9.16) por Apache Lucene, para que pueda ser indexado por esta herramienta.

```
Document doc = this.getTextExtractor().doGetDocument(param);
Reader rrr = doc.getField("contents").readerValue();

//OBTENER LA EXTENSION DEL ARCHIVO
String cpath = file.getCanonicalPath();

 int index=-1;
 String extension = "NO_EXT";
 if( (index=cpath.lastIndexOf("."))!=-1 )
      extension = cpath.substring(index+1);

String summary = doc.getField("summary") == null?null: doc.getField("summary").stringValue();
if (summary == null)
    summary = path;

int titleSize = Math.min(50, summary.length());
String title = (doc.getField("title") == null?null:doc.getField("title").stringValue());
if (title == null || "".equals(title))
    title = summary.substring(0, titleSize);

Resource r = ((InternalCompassSession) session).getCompass().getResourceFactory()
                                                    .createResource(ESBConstants.FS_ALIAS_KEY);
r.addProperty("uid", "uid-" + path);
r.addProperty("creator",  doc.getField("Author") == null?ESBConstants.NO_AUTOR_MSG:doc.getField("Autho
r.addProperty("datasource", this.getName());
r.addProperty("url", path);
r.addProperty("modified", file.lastModified());
r.addProperty("creationdate", doc.getField("creationDate") == null?"": doc.getField("creationDate"));
r.addProperty("type", extension);
r.addProperty("title", title.trim());
r.addProperty("keywords",  doc.getField("keywords") == null?"":  doc.getField("keywords"));
r.addProperty("summary", summary.trim());
r.addProperty("contents", rrr);

session.create(r);
```

FIGURA 9.16 – Transformación de los datos extraídos a formato de documento entendible por Lucene

En un archivo XML externo (FIGURA 9.17) se realiza la configuración de los diferentes campos que conforman el documento Lucene.

```
<resource alias="FS">

      <resource-id name="uid" />
      <resource-property name="creator" store="yes" index="un_tokenized"/>
      <resource-property name="contents" store="no" index="tokenized"/>
      <resource-property name="summary" store="yes" index="no"/>
      <resource-property name="url" store="yes" index="un_tokenized"/>
      <resource-property name="type" store="yes" index="un_tokenized"/>
      <resource-property name="datasource" store="yes" index="un_tokenized"/>
      <resource-property name="modified" store="yes" index="un_tokenized"/>
      <resource-property name="creationdate" store="yes" index="un_tokenized"/>
      <resource-property name="keywords" store="yes" index="un_tokenized"/>
      <resource-property name="title" store="yes" index="un_tokenized"/>

 </resource>
```

FIGURA 9.17 – Configuración de los campos a ser indexados por Apache Lucene

El adaptador al file system se configura declarativamente en archivos XML externos (FIGURA 9.18 y 9.19) utilizando las facilidades brindadas por el framework Spring

```
<bean abstract="true" id="fsAdapter" class="ar.edu.tesis.adapters.impl.FSAdapterImpl">
    <property name="textExtractor" ref="esb-text-extractor" />
</bean>
```

FIGURA 9.18 - Extracto de configuración de un archivo XML del framework

```
<bean id="fsDevice" parent="fsAdapter">
        <property name="name"><value>fsAdapter</value></property>
        <property name="url"><value>/usr/local/apache-servicemix-3.2.2/fsadapter</value></property>
        <property name="compass"><ref local="localCompass" /></property>
</bean>
```

FIGURA 9.19 - Extracto de configuración de un archivo XML del caso de estudio

B. Servicio de Búsqueda

El proceso de búsqueda implica la recepción, análisis y procesamiento del query (FIGURA 9.20) enviado por el cliente (dependiendo el escenario puede ser un cliente humano o máquina), y la ejecución federalizada de la búsqueda en todas las fuentes de información registradas.

```
//SE CALCULA EL TIEMPO DE BUSQUEDA
StopWatch stopW = new StopWatch();
stopW.start();

//SE PROCESA EL QUERY DE ENTRADA
String q = query.getSearchQuery();

CompassQuery luceneQuery;

luceneQuery = session.queryBuilder()

    .queryString(q)
    .toQuery();

 CompassHits hitsSort;

 String orderBy = query.getOrderBy();
 if (null == orderBy || "".equals(orderBy) || "default".equalsIgnoreCase(orderBy))
     hitsSort = luceneQuery.hits();

 else
     hitsSort = luceneQuery
        .addSort(query.getOrderBy(), CompassQuery.SortPropertyType.AUTO)
        .addSort(query.getOrderBy(), CompassQuery.SortDirection.AUTO)
        .hits();



//PAGINACION ************************************
Integer from = query.getFrom();
Integer to = query.getTo();
```

FIGURA 9.20 - Extracto del código del servicio de búsqueda que procesa el query recibido

Una vez obtenidos los resultados desde el índice, se realiza la transformación desde el formato propio de Apache Lucene, a una estructura de objetos específica del caso de estudio (FIGURA 9.21).

El proceso de transformación del resultado al formato requerido por el cliente es delegado en otros servicios.

```
sr = new SearchResult();

sr.setAlias(hit.getAlias());

Property uid = hit.getResource().getIdProperty();
sr.setUid((uid == null?ESBConstants.NO_UID:uid.getStringValue()));

Property ds = hit.getResource().getProperty("datasource");
sr.setDataSource((ds == null?hit.getAlias():ds.getStringValue()));

sr.setScore(hit.getScore());

Property autor = hit.getResource().getProperty("creator");
sr.setCreator((autor == null?ESBConstants.NO_AUTOR_MSG: autor.getStringValue()));

Property type = hit.getResource().getProperty("type");
sr.setType((type == null?ESBConstants.NO_EXTENSION: type.getStringValue()));

Property url = hit.getResource().getProperty("url");
sr.setUrl((url == null?"#":url.getStringValue()));

Property sProperty = hit.getResource().getProperty("summary");
String summaryWH =(sProperty == null)?"":sProperty.getStringValue();

String summary = hit.getHighlightedText().getHighlightedText("summary");

sr.setSummary(summary == null?summaryWH:summary);

Property pTitle = hit.getResource().getProperty("title");
```

FIGURA 9.21 – Transformación del formato de documento Lucene a un formato interno requerido por el caso de estudio

Igualmente como sucede con el proceso de indexación, el servicio de búsqueda necesita conocer los adaptadores a las herramientas colaborativas asincrónicas, a fin de poder interactuar con éstas.

Es necesario configurar el servicio con los adaptadores implementados (FIGURA 9.22).

Para ésto se hace uso de las facilidades provistas por el motor DI (*Dependency Injection*) del framework Spring

```
<bean id="localSearchContainer" parent="searchContainer">

      <property name="compass"><ref local="localCompass" /></property>

      <property name="gpsDevices">
        <list>
           <ref local="mantisDevice" />
           <ref local="mediaWikiDevice" />
           <ref local="svnGpsDevice" />
           <ref local="fsDevice"/>

        </list>
      </property>

</bean>

   <bean id="localSearcher" class="ar.edu.tesis.search.DefaultESBSearcher">
        <property name="compass" ref="localCompass"/>
        <property name="container" ref="localSearchContainer"/>
    </bean>
```

FIGURA 9.22 - Extracto de configuración declarativa del servicio de búsqueda

9.5. Servicio de búsqueda basada en texto simple sobre HTTP

Este servicio se encarga de procesar los parámetros enviados por el cliente del escenario de búsqueda basado en texto simple, y realizar la configuración interna que jESBihca requiere para ejecutar la búsqueda en el índice Lucene.

Depende únicamente de una implementación del motor de búsqueda, inyectado utilizando un archivo de configuración externo (FIGURA 9.23), basado en el framework Spring

```
<import resource="classpath:search-engine-context.xml" />

<bean:endpoint service="esb:searchingSimple" endpoint="searchSimpleEndpoint" bean="#searchSimple"/>

<bean id="searchSimple" class="ar.edu.tesis.usecase.searching.SearchSimple">
  <property name="searcher" ref="localSearcher" />
</bean>
```

FIGURA 9.23 - Extracto de la configuración declarativa del servicio de búsqueda de texto plano

Luego de realizar la búsqueda, procesa los resultados, los convierte en XML, formato requerido y entendible por el router de *jESBihca*, y posteriormente envía el mensaje normalizado nuevamente al ESB para que sea atendido por el servicio destinatario (FIGURA 9.24).

```
public void onMessageExchange(MessageExchange exchange) throws MessagingException {

    InOut in = (InOut) exchange;

     if (in.getStatus() == ExchangeStatus.DONE) {
            return;}
               else if (in.getStatus() == ExchangeStatus.ERROR) {return;}
    try {

        NormalizedMessage nm = in.getInMessage();
        String orderBy = (String) nm.getProperty("orderBy");

        SearchQuery sq = toSearchQuery(nm);
        sq.setOrderBy(orderBy);
        SearchResults srs = this.doSearch(sq);

        StringSource ss = toStringSource(srs);

        nm.setContent(ss);

        in.setOutMessage(nm);

    } catch (Exception e) {
        in.setError(e);

    } finally {
        channel.send(in);
    }
```

FIGURA 9.24 - Extracto de código del servicio de búsqueda de texto plano

En el escenario de búsqueda simple, el cliente humano espera recibir los resultados en formato HTML, por lo tanto previo a retornar los resultados al cliente, éstos se envían al servicio de transformación basado en plantillas XSL (FIGURA 9.25), para que los transforme al formato adecuado. Luego de la transformación se envían los resultados al cliente.

```
.choice()
    .when(xpath("//searchType='simple'"))
        .to(SEARCH_AUDIT + "?mep=in-out")
        .convertBodyTo(DOMSource.class)
        .to(SEARCH_SIMPLE_OUT + "?mep=in-out")
         .to(SEARCH_AUDIT + "?mep=in-out")
        .to(SEARCH_XSLT_OUT + "?mep=in-out")

    .when(xpath("//searchType='xml'"))
```

FIGURA 9.25 - Extracto de código del router de jESBihca, donde redirecciona al servicio XSLT antes de retornar el resultado al servicio HTTP del escenario simple

Para el acceso desde los clientes este escenario implementa un servicio conector (del tipo BC según la especificación JBI) para atender requerimientos HTTP (FIGURA 9.26).

Este servicio se encarga de analizar los parámetros recibidos desde el formulario cliente y enviarlos a jESBihca para su procesamiento, transformación y enrutamiento.

```
public MessageExchange createExchange(HttpServletRequest request, ComponentContext context) throws Exceptio

    MessageExchange me = context.getDeliveryChannel().createExchangeFactory().createExchange(getDefaultMep(
    NormalizedMessage in = me.createMessage();

    ServletInputStream sis = request.getInputStream();
    Map parameters = request.getParameterMap();

    String[] pReq = (String[]) parameters.get("request");
    String[] pMaxResults = (String[]) parameters.get("maxResults");
    String[] pCheckAudit = (String[]) parameters.get("checkAudit");
    String[] pCheckLog = (String[]) parameters.get("checkLog");
    String[] pUsername = (String[]) parameters.get("username");
    String[] pOrderBy = (String[]) parameters.get("orderBy");
    String[] pSort = (String[]) parameters.get("sort");
    String[] pFrom = (String[]) parameters.get("pagFrom");
    String[] pTo = (String[]) parameters.get("pagTo");

    boolean bCheckAudit = (pCheckAudit == null?false:Boolean.parseBoolean(pCheckAudit[0]));
    boolean bCheckLog = (pCheckLog == null?false:Boolean.getBoolean(pCheckLog[0]));
    String username = (pUsername == null)?"NO_USERNAME":pUsername[0];
    String orderBy = (pOrderBy == null || "".equals(pOrderBy))?"DEFAULT":pOrderBy[0];
    String sort = (pSort == null)?"ASC": pSort[0];
    int from = (pFrom == null)?1:Integer.parseInt(pFrom[0]);
    int to = (pTo == null)?10:Integer.parseInt(pTo[0]);
    String xmlout = "<search><searchQuery>" + pReq[0] + "</searchQuery>";
    xmlout += "<maxResults>" + 10 + "</maxResults><from>" + from + "</from><to>" + to + "</to></search>";

    in.setProperty("checkAudit", new Boolean(bCheckAudit));
    in.setProperty("checkLog", new Boolean(bCheckLog));
    in.setProperty("username", username);
    in.setProperty("orderBy", orderBy);
    in.setProperty("sort", sort);
```

FIGURA 9.26 - Extracto del código que recibe los parámetros de entrada enviados vía HTTP por el cliente de la búsqueda simple

jESBihca retorna los resultados en formato HTML y este servicio se encarga de responder al cliente (FIGURA 9.27)

```
public void sendOut(MessageExchange exchange, NormalizedMessage outMsg, HttpServletRequest request,
        HttpServletResponse response) throws Exception
    {

    NormalizedMessage nm = exchange.getMessage("out");
    SourceTransformer transformer = new SourceTransformer();
    String res = transformer.contentToString(nm);

    String mimeType  = (String) nm.getProperty("mimeType");


    try
        {

        ServletOutputStream sos = response.getOutputStream();
        response.setStatus(HttpServletResponse.SC_OK);
        sos.write(res.getBytes());
        sos.close();
        }
    catch (Exception e)
        {
              logger.log(Level.SEVERE, "EXCEPCION EN SEARCHSIMPLEHTTPMARSHALER: " + e.getMessage(), e);
              e.printStackTrace();
        }

    }
```

FIGURA 9.27 - Extracto de código que envía la respuesta en formato XML al cliente

9.6. Servicio de búsqueda basada en documentos XML sobre HTTP

Un servicio HTTP es básicamente un servicio accesible a través del protocolo HTTP. En Java, una de las formas de implementar tal servicio es a partir del uso de Servlets, para ser desplegado en un contenedor de servlets/JSP, tal es el caso de Apache Tomcat o Jetty.

En el caso de estudio implementado este servicio HTTP recibe y retorna documentos XML por lo tanto lo denominaremos **Servicio XML.**

Para la implementación de Servicios HTTP que reciban y retornen documentos XML con una estructura fija predeterminada, fue diseñado e implementado un ***framework*** simple que facilite su construcción (FIGURA 9.28).

El *framework* está compuesto por las siguientes clases, enumeradas con una breve descripción de su función:

- **FmkXMLServiceXMLMessageBroker:** broker de mensajes que funciona como el punto de entrada que instancia los demás componentes del framework, a partir de parámetros "inyectados" a través de archivos de configuración externos, basados en Spring.
- **FmkXMLServiceFactory:** superclase de las "fábricas" de instancias.

- **FmkXMLServiceRequest:** es la representación interna del pedido entrante.
- **FmkXMLServiceRequestHandler:** manejador del pedido entrante
- **FmkXMLServiceRequestHandlerFactory:** "fábrica" de manejadores
- **FmkXMLServiceResponse:** representación interna de la respuesta del servicio XML
- **FmkXMLServiceXMLParser**: implementación interna del *parser* utilizado para analizar documentos XML
- **FmkXMLServiceXMLRequestParserFactory**: "fábrica" de analizadores (*parsers*) de documentos XML
- **FmkXMLServiceXMLResponseComposer:** implementación del componente que construye el documento XML respuesta
- **FmkXMLServiceXMLResponseComposerFactory**: "fábrica" de los componentes "*Composers*"
- **FmkXMLServiceErrorResponse**: encapsula el error producido al acceder al servicio XML

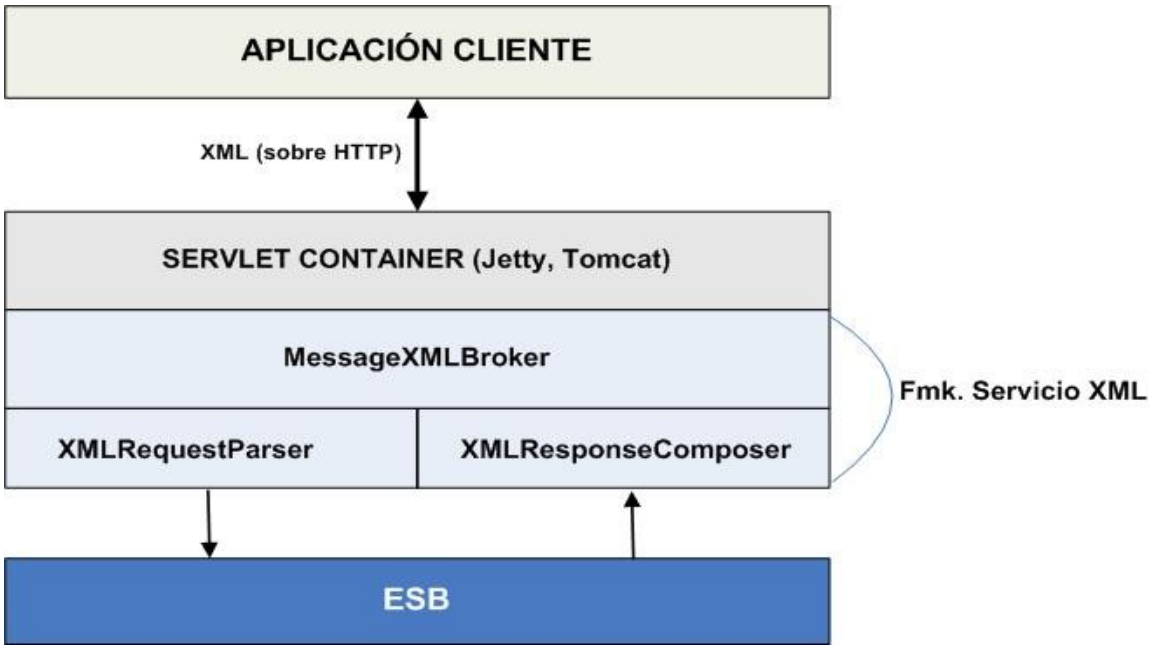


FIGURA 9.28 - Principales componentes del framework Servicios XML

9.6.1. Interacción entre los componentes

El cliente envía el pedido (*request*) que encapsula el documento XML con una estructura prefijada. El envío se hace a través de HTTP, por lo tanto es necesario que exista un servicio (FIGURA 9.29) que atienda requerimientos HTTP desplegado en jESBihca.

El contenedor de servlets/JSP, Jetty en el caso de estudio implementado, recibe el pedido en un puerto predeterminado, y lo reenvía al servicio HTTP que concuerde con la URL ingresada

```
<beans xmlns:http="http://servicemix.apache.org/http/1.0"
       xmlns:esb="http://tesis.edu.ar/esb">

   <http:consumer
                    service="esb:httpsearchxml"
                    endpoint="endpoint"
                    targetService="esb:search-xml-routingSlip"
                    locationURI="http://localhost:8193/searchxml/"
                    defaultMep="http://www.w3.org/2004/08/wsdl/in-out"
                    />


</beans>
```

FIGURA 9.29 – Configuración del servicio HTTP que recibe los documentos XML de entrada

El servicio HTTP deriva el pedido al servicio manejador del *broker* implementado, *DefaultXMLServiceMessageBroker*

La FIGURA 9.30 muestra el extracto de código del servicio que configura el *broker* implementado

```
<?xml version="1.0" encoding="UTF-8"?>

<beans>

  <import resource="classpath:abstract-broker-context.xml" />
  <import resource="classpath:search-engine-context.xml" />

  <bean id="usecase-xmlservice-defaultbroker" parent="xmlservice-defaultbroker"
        class="ar.edu.tesis.usecase.service.xml.DefaultXMLServiceMessageBroker">

     <property name="applicationPackage"><value>ar.edu.tesis.usecase.service.xml</value></property>
     <property name="suffixXMLRequestParser"><value>XMLRequestParser</value></property>
     <property name="suffixRequestHandler"><value>RequestHandler</value></property>
     <property name="suffixXMLResponseComposer"><value>XMLResponseComposer</value></property>
     <property name="searcher" ref ="localSearcher" />

  </bean>

</beans>
```

FIGURA 9.30 – Extracto de la configuración del broker del servicio XML implementado

Para que la implementación del *broker* pueda interactuar con el contenedor JBI (Apache ServiceMix), es necesario configurarlo como un servicio compatible con el estándar JBI (FIGURA 9.31). De esta manera será capaz de recibir y enviar mensajes a otros servicios desplegados en jESBihca.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:bean="http://servicemix.apache.org/bean/1.0"
       xmlns:esb="http://tesis.edu.ar/esb">

    <import resource="classpath:defaultbroker-context.xml" />

    <bean:endpoint service="esb:searchingXML"
                   endpoint="searchXMLEndpoint"
                   bean="#usecase-xmlservice-defaultbroker"/>

</beans>
```

FIGURA 9.31 – Configuración del broker del servicio XML como servicio JBI compatible

El manejador del *broker* invoca la funcionalidad (FIGURA 9.32) que pone en marcha los componentes del *framework*.

```
public String handleRequest( String xmlRequest ) {

    FmkXMLServiceXMLRequestParser parser = null;
    FmkXMLServiceRequest request = null;
    FmkXMLServiceRequestHandler handler = null;
    FmkXMLServiceResponse response = null;

    String rootNodeName = "_Nombre Nodo Raiz Invalido_";

   try {
        rootNodeName = FmkXMLServiceXMLParser.getRootNodeName( xmlRequest );
    }catch ( Exception e ) {
        e.printStackTrace();
    }

    try {

        parser = parserFactory.newInstance( rootNodeName );

        request = parser.parse( xmlRequest );

        request.setSearcher(searcher);

        handler = handlerFactory.newInstance( request );

        response = handler.handle( request );

    } catch ( Exception e ) {

        e.printStackTrace();
        response = getExceptionResponse( e );
    }
```

FIGURA 9.32 - Extracto de código del método encargado de delegar responsabilidades

El primer componente instanciado es el *parser* del documento XML. La clase que implementa el *parser* tiene como sufijo *XMLRequestParser*. En particular en el caso de estudio implementado se ha usado un *parser* SAX[66]

Con la información contenida en el documento XML, el parser construye un objeto *Request*, utilizado para instanciar un objeto manejador, quien se encargará de invocar el servicio de búsqueda.

La FIGURA 9.33 muestra un extracto simplificado del código perteneciente al manejador del *Request* donde se realiza la invocación al servicio de búsqueda

66 **SAX parser:** http://saxproject.org

```
private FmkXMLServiceResponse handleSearchRequest(SearchRequest request) {

    SearchResponse response = new SearchResponse();

    SearchQueryImpl criteria = new SearchQueryImpl();

    try {

        response.setTypeName(request.getTypeName());

        if (request.getQuery() != null)
            criteria.setQuery(request.getQuery());

        if (request.getMaxResults() != null)
            criteria.setMaxResults(request.getMaxResults());

        SearchResults results = request.getSearcher().doSearch(criteria);

        response.setSearchResults(results);

        return response;


    } catch (Exception e) {
        e.printStackTrace();

    }

    return response;

}
```

FIGURA 9.33 – Extracto de código donde se invoca el servicio de búsqueda desde el servicio XML

El código anterior construye y devuelve la respuesta (*Response*) que encapsula los datos retornados por el servicio de búsqueda.

9.6.2. Diagrama de Interacción

El diagrama 9.34 muestra gráficamente la interacción entre los componentes de una instancia del Servicio XML. Es necesario notar que la convención en la estructura del documento XML e implementación del *framework* debe respetarse.

Es así que nodo raíz del documento XML coincide con el prefijo usado en todas las clases que componen la implementación del *framework*. De esta manera documentos XML con distintos nodos raíces invocarán dinámicamente distintas instancias del servicio XML.

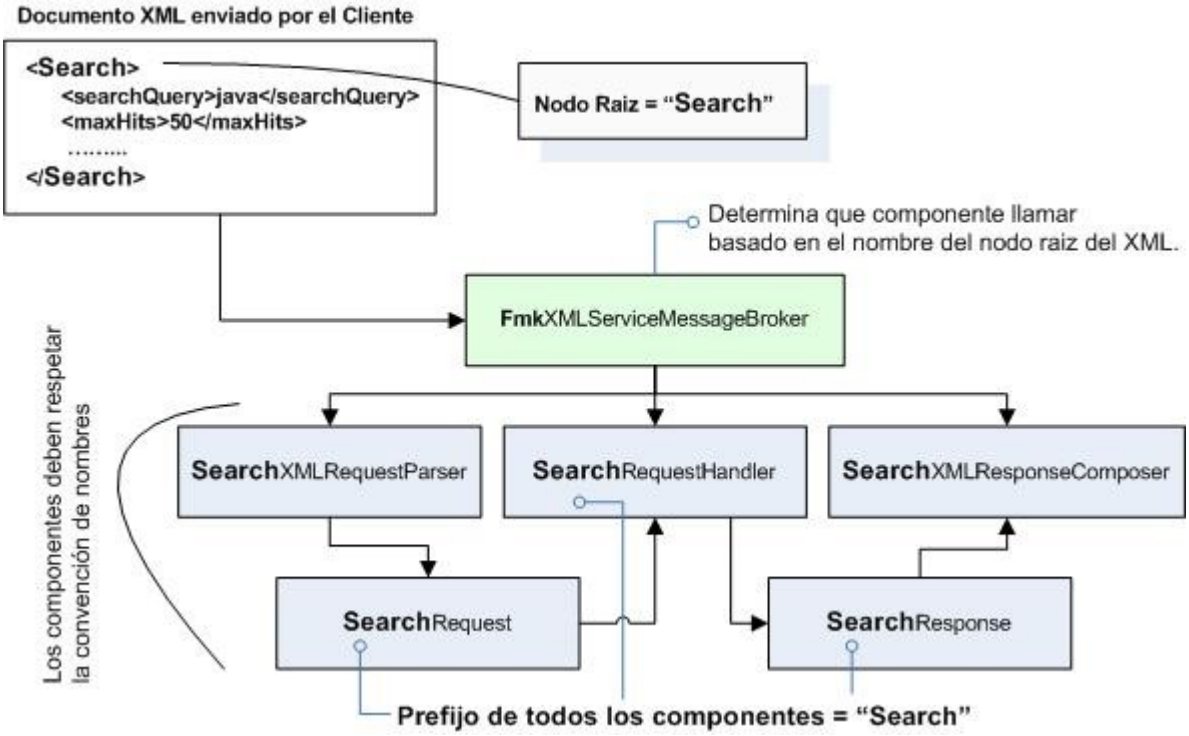


FIGURA 9.34 – Interacción entre componentes del Servicio XML implementado

A. Convención de uso del Servicio XML

Dado un documento XML con nodo raíz = "**Search**", entonces

1. Los componentes deben respetar las siguientes convenciones de nombre

- **<Search>**XMLRequestParser
- **<Search>**RequestHandler
- **<Search>**XMLResponseComposer

La FIGURA 9.35 muestra la estructura de paquetes correspondiente al servicio XML implementado para el caso de estudio, donde se puede observar como cada clase implementada respeta la convención requerida por el framework de servicios XML.

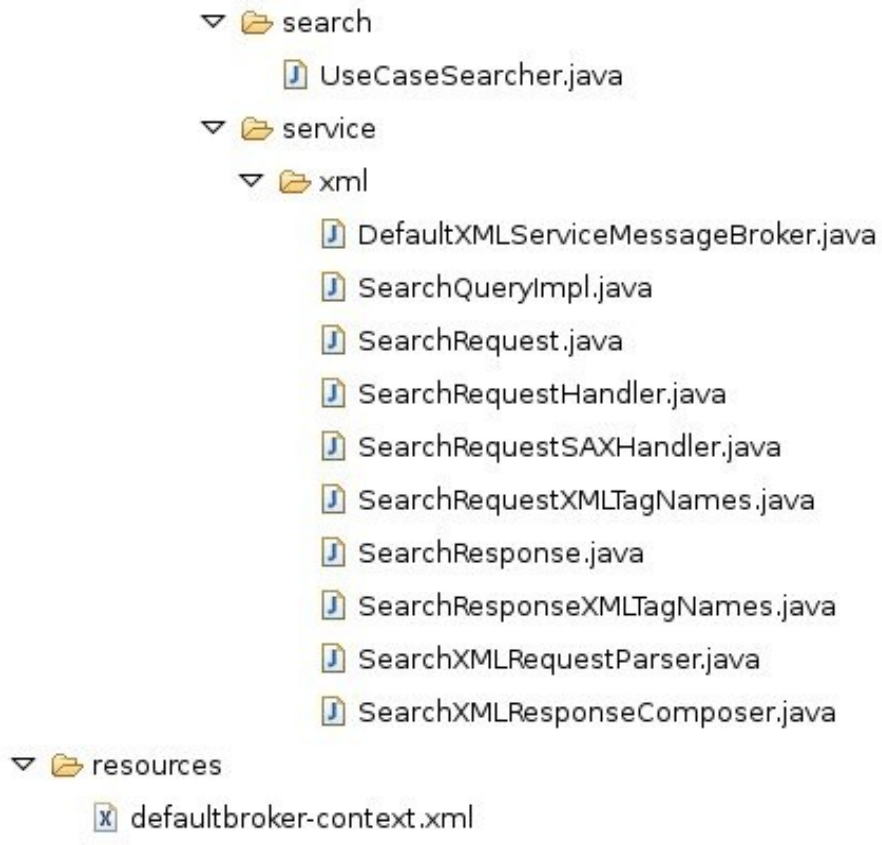


FIGURA 9.35 – Estructura de directorios del Servicio XML implementado

2. Deben implementar las interfaces correctas

- FmkXMLServiceRequestParser
- FmkXMLServiceRequest
- FmkXMLServiceRequestHandler
- FmkXMLServiceResponse
- FmkXMLServiceErrorResponse
- FmkXMLServiceResponseComposer

La FIGURA 9.36 muestra la estructura de paquetes del framework para implementar Servicios XML, donde se observan las interfaces enumeradas anteriormente

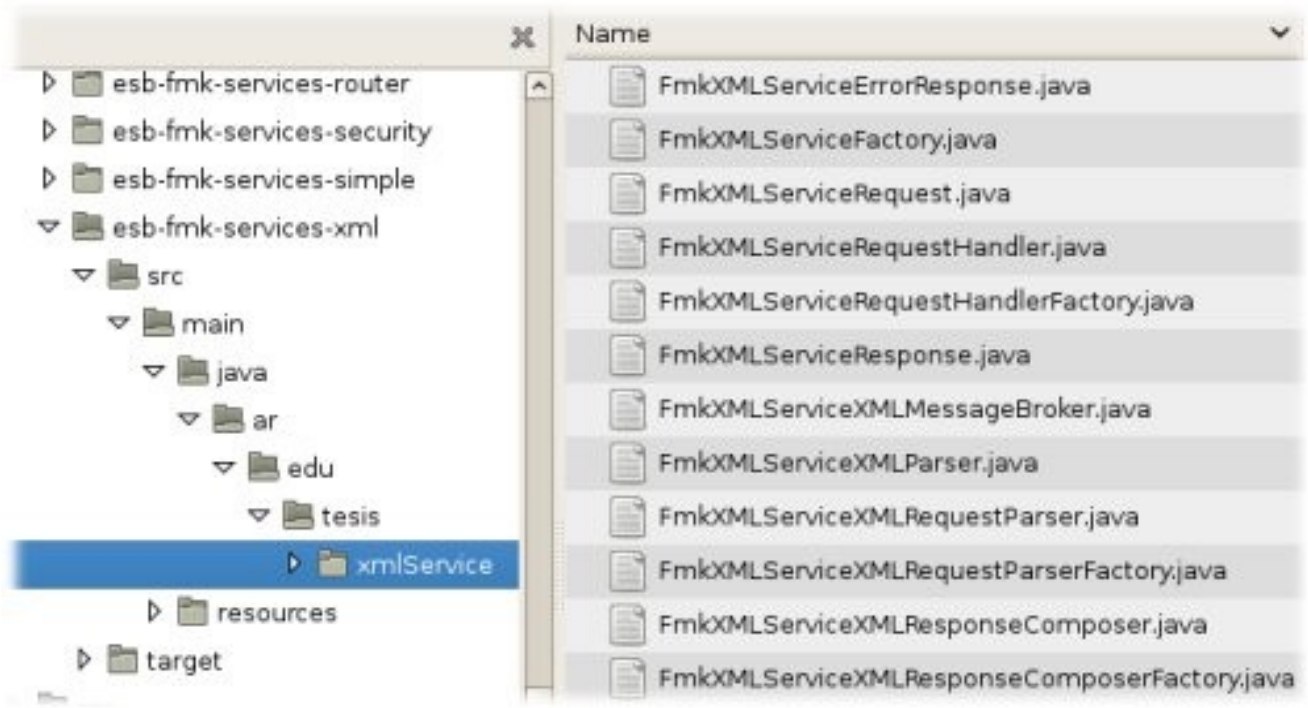


FIGURA 9.36 – Estructura de paquetes del framework para crear servicios XML

3. Respetando estas convenciones, los componentes son automáticamente "capturados" por la implementación del *Message Broker* (FIGURA 9.37)

```
public class DefaultXMLServiceMessageBroker extends FmkXMLServiceXMLMessageBroker implements MessageExchangeListener {

    private static final Log logger = LogFactory.getLog(DefaultXMLServiceMessageBroker.class);

    @Resource
    private DeliveryChannel channel;


    public DefaultXMLServiceMessageBroker() {
        this("xmlservice");

    }

    public DefaultXMLServiceMessageBroker(String _applicationName) {
           super(_applicationName);


    }
```

FIGURA 9.37 – Extracto del código del broker del Servicio XML implementado

9.7. Servicio de búsqueda basada en mensajes SOAP

Este servicio es necesario para clientes máquinas que requieran realizar búsquedas en las diferentes herramientas colaborativas de una manera automatizada y basada en estándares.

Se compone de 2 servicios básicos, de acuerdo a la especificación JBI: un servicio BC (*Binding Component*), y un servicio SE (*Service Engine*). Estos servicios, básicamente, son implementados utilizando la librería Apache CXF, y *plugins* de Apache Maven para la generación de código Java a partir de documentos WSDL (FIGURA 9.38)

```
<plugin>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-codegen-plugin</artifactId>
    <version>${cxf-version}</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <configuration>
                <sourceRoot>${basedir}/target/jaxws</sourceRoot>
                <wsdlOptions>
                    <wsdlOption>
                        <wsdl>${basedir}/src/main/resources/wsdl-search.wsdl</wsdl>
                        <extraargs>
                            <extraarg>-verbose</extraarg>
                        </extraargs>
                    </wsdlOption>

                </wsdlOptions>

            </configuration>
            <goals>
                <goal>wsdl2java</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

FIGURA 9.38 - Extracto del archivo de configuración de Apache Maven del servicio SOAP SE

El servicio BC es configurado a partir de un archivo XML (FIGURA 9.39), se encarga de recibir y procesar el requerimiento por parte del cliente (máquina), y reenviarlo al servicio destino previamente configurado.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:esb="http://tesis.edu.ar/usecase/search">


<cxfbc:consumer
    useJBIWrapper="false"
    wsdl="classpath:wsdl-search.wsdl"
    targetService="esb:soap-routingSlip"/>

</beans>
```

FIGURA 9.39 – Configuración del servicio SOAP BC

En el caso de estudio planteado el servicio destino para el servicio SOAP BC es un servicio que implementa el patrón Routing Slip (ver Capítulo 2, sección 2.8.2.B).

Este servicio, al igual que la mayoría de los servicios implementados en el caso de estudio, es configurado declarativamente mediante un archivo XML (FIGURA 9.40)

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:esb="http://tesis.edu.ar/usecase/search"
       xmlns:xslt="http://tesis.edu.ar/esb">


<eip:static-routing-slip service="esb:soap-routingSlip" endpoint="endpoint">

    <eip:targets>

        <eip:exchange-target service="esb:DoSearchService" interface="esb:Search"/>
        <eip:exchange-target service="xslt:xsltSoap" endpoint="xslt:xsltSoapEndpoint"/>

    </eip:targets>

</eip:static-routing-slip>


</beans>
```

FIGURA 9.40 – Configuración del servicio EIP para mensajes SOAP

El servicio SOAP SE (FIGURA 9.41) recibe los parámetros enviados desde la aplicación cliente, los procesa e invoca la búsqueda federalizada a través del motor de búsqueda full text. El resultado de la búsqueda es enviado a jESBihca para que sea retornado al cliente.

```
package ar.edu.tesis.usecase.search;

import java.util.Iterator;
import javax.jws.WebService;
import javax.xml.ws.Holder;
import ar.edu.tesis.search.*;
import ar.edu.tesis.search.DefaultESBSearcher;
import ar.edu.tesis.usecase.search.types.DoSearch;
import ar.edu.tesis.usecase.search.types.DoSearchResponse;

@WebService(serviceName = "DoSearchService",
            targetNamespace = "http://tesis.edu.ar/usecase/search",
            endpointInterface = "ar.edu.tesis.usecase.search.Search")
public class DoSearchImpl extends DefaultESBSearcher implements Search {

    public ar.edu.tesis.usecase.search.types.DoSearchResults doSearch(
                    String searchQuery, int to, int from, String username,
                    boolean audit, boolean log, String orderBy)
                                        throws UnknownDoSearchFault {

        if (searchQuery == null){
            ar.edu.tesis.usecase.search.types.UnknownDoSearchFault f =
                 new ar.edu.tesis.usecase.search.types.UnknownDoSearchFault();
            f.setMessage("query NULL");
            throw new UnknownDoSearchFault(null, f);
        }

        SearchQuery query = new SearchQuery();
        query.setSearchQuery(searchQuery);
```

FIGURA 9.41 - Extracto del código de implementación del servicio SOAP SE

Este servicio es configurado mediante un archivo XML de manera declarativa (FIGURA 9.42)

```
<import resource="classpath:search-engine-context.xml" />

 <cxfse:endpoint useJBIWrapper="false">

     <cxfse:pojo>
       <bean class="ar.edu.tesis.usecase.search.DoSearchImpl">
            <property name="compass" ref="localCompass" />
            <property name="container" ref="localSearchContainer" />
       </bean>
     </cxfse:pojo>

     <cxfse:inInterceptors>
         <bean class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
     </cxfse:inInterceptors>
     <cxfse:outInterceptors>
         <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>
     </cxfse:outInterceptors>
     <cxfse:inFaultInterceptors>
              <bean class="org.apache.cxf.interceptor.LoggingInInterceptor"/>
      </cxfse:inFaultInterceptors>
      <cxfse:outFaultInterceptors>
         <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor"/>
      </cxfse:outFaultInterceptors>

 </cxfse:endpoint>
```

FIGURA 9.42 - Extracto del archivo XML de configuración del servicio SOAP SE

Para facilitar las pruebas de este escenario, se ha implementado un cliente web que simula ser una máquina al enviar mensajes en formato SOAP. Al ser un cliente web es necesario transformar el mensaje SOAP de respuesta a formato HTML para su mejor visualización. Es por ésto que se ha codificado un servicio de transformación basado en plantillas XSL (FIGURA 9.43) que transforma el resultado de la búsqueda de formato SOAP a formato HTML.

```
<xsl:template match="/">
   <table align="center" width="700px" border="0" cellpadding="1" cellspacing="0">
                     <xsl:apply-templates select="//soap:Body" />
   </table>
</xsl:template>

<xsl:template match="//soap:Body">
    <xsl:apply-templates select="//soap:Body/esb:DoSearchResponse/*"  />
</xsl:template>

<xsl:template match="esb:searchSummary">
 <tr><td colspan="4" align="center">
          <table class="tablePropiedadesSummary">
             <tr>
               <td><label class="lbl">Resultados mostrados:</label>
                    <label class="lblPropiedades"><xsl:value-of select="esb:resultadosParcial"/></label>|</td>
               <td><label class="lbl">Numero resultados totales:</label>
                    <label class="lblPropiedades"><xsl:value-of select="esb:resultadosTotal"/></label>|</td>
               <td><label class="lbl">Tiempo de busqueda:</label>
                    <label class="lblPropiedades"><xsl:value-of select="esb:searchTime"/>ms </label></td>
             </tr>
          </table>
   </td> </tr>
</xsl:template>

<xsl:template match="esb:response">

<xsl:apply-templates select="esb:searchSummary"/>
```

FIGURA 9.43 - Extracto de la plantilla XSL de transformación de mensajes SOAP a HTML

Estos 3 *service units* (denominación usada en la especificación JBI), SOAP BC, SOAP EIP y SOAP SE, son relacionados y empaquetados en un *service assembly* (denominación usada en la especificación JBI), y desplegados en el contenedor JBI.

La FIGURA 9.44 muestra un extracto de la configuración del service assembly que implementa el servicio SOAP

Esta es la forma requerida por la especificación JBI para el despliegue de servicios.

```
<?xml version="1.0" encoding="UTF-8"?>
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <service-assembly>
    <identification>
      <name>esbusecase-consumers-search-soap-sa</name>
      <description>ESB : CONSUMERS : SEARCH SOAP : CXF - SA</description>
    </identification>
    <service-unit>
      <identification>
        <name>esbusecase-consumers-search-soap-eip-su</name>
        <description>ESB :: USECASE</description>
      </identification>
      <target>
        <artifacts-zip>esbusecase-consumers-search-soap-eip-su-1.0.zip</artifacts-zip>
        <component-name>servicemix-eip</component-name>
      </target>
    </service-unit>
    <service-unit>
      <identification>
        <name>esbusecase-consumers-search-soap-bc-su</name>
        <description>ESB :: USECASE</description>
      </identification>
      <target>
        <artifacts-zip>esbusecase-consumers-search-soap-bc-su-1.0.zip</artifacts-zip>
        <component-name>servicemix-cxf-bc</component-name>
      </target>
    </service-unit>
  </service-assembly>
</jbi>
```

FIGURA 9.44 – Configuración del Service Assembly para el servicio SOAP

9.8. Servicios que implementan Patrones EAI

Apache ServiceMix provee un componente JBI, ***servicemix-eip***, que facilita la implementación de los patrones EAI (*Enterprise Application Integration*) discutidos y analizados en el capítulo 2, sección 2.8.2 titulada "Patrones de Diseño en Integración"

En la implementación de algunos patrones se hace uso de la librería Apache Camel para facilitar aspectos relacionados con ruteo o mediación de mensajes.

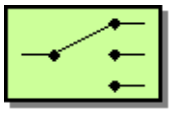
Los principales patrones EAI utilizados e implementados en el caso de estudio se corresponden con:

- Patrón Content-based router
- Patrón Content enricher
- Patrón Static recipient list
- Patrón Message Filter
- Patrón Static routing slip
- Patrón Message Translator

A. Servicio basado en el patrón Content-based Router

Este patrón fue analizado anteriormente como un componente fundamental para la implementación de un ESB. Básicamente, este tipo de *router* consume un mensaje desde un canal, y basado en un conjunto de condiciones del *header* o contenido del *body* del mensaje, lo republica en un canal de mensajes diferente

Notación

 (imagen extraída de [87])

Implementación en el caso de estudio

Componentes involucrados:

- **Cliente**: cliente que envía mensajes que serán transformados a un formato XML canónico por el ESB. Estos mensajes tienen como destinatario final el servicio de búsqueda e indexación.
- **Componente Content-based Router:** servicio desplegado en jESBihca e implementado a través de la librería Apache Camel (FIGURA 9.45). En base al contenido del mensaje XML que recibe lo rutea, sin alterarlo, a uno de los componentes receptores configurados. En particular, en el caso de estudio es reenviado al servicio de búsqueda e indexación, y auditoría (en caso que corresponda).

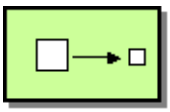
- **Receptor**: el receptor para este flujo de mensajes es un componente que extrae la información que requiere el servicio de búsqueda para realizar la búsqueda en las diferentes fuentes de información registradas. Invoca el servicio de búsqueda y el resultado es retornado al cliente.

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:camel="http://activemq.apache.org/camel/schema/spring"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://activemq.apache.org/camel/schema/spring
       http://activemq.apache.org/camel/schema/spring/camel-spring.xsd">

    <import resource="log-bd.xml"/>

    <camel:camelContext id="router"  xmlns="http://activemq.apache.org/camel/schema/spring">

        <package>ar.edu.tesis.usecase.services</package>
        <camel:jmxAgent id="agent" createConnector="true"/>

    </camel:camelContext>

    <camel:endpoint id="logStore" uri="file:///home/cristian/testcamel/"/>
    <bean id="logProcessor" class="ar.edu.tesis.usecase.services.LogProcessor" />

</beans>
```

FIGURA 9.45 - Extracto de la configuración del router basado en Apache Camel

B. Servicio basado en el patrón Content Enricher

Notación

 (imagen extraída de [87])

Implementación en el caso de estudio

Componentes involucrados:

- **Cliente**: cliente que envía mensajes que serán transformados a un formato XML canónico por el ESB. Estos mensajes tienen como destinatario final el servicio de búsqueda e indexación.
- **Componente Content enricher:** servicio desplegado en el componente servicemix-eip. Enriquece el mensaje original con información adicional. El mensaje enriquecido es enviado a un componente receptor. En el caso de

estudio implementado, este componente es utilizado en diferentes situaciones. En el escenario de búsqueda simple se utiliza este patrón (FIGURA 9.46) para enriquecer el mensaje entrante con información requerida por el componente de enrutamiento (FIGURA 9.47).

```
<beans xmlns:saxon="http://servicemix.apache.org/saxon/1.0"
       xmlns:esb="http://tesis.edu.ar/esb">

  <saxon:xslt service="esb:enricherSearchSimpleXslt"
              endpoint="enricherSearchSimpleEndpoint"
              resource="classpath:transform.xsl" />

</beans>
```

FIGURA 9.46 – Configuración del servicio que implementa el patrón Content Enricher

```
<xsl:stylesheet
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='2.0'>

  <xsl:output method="xml" indent="yes" encoding="UTF-8"/>

  <xsl:template match="/">

      <search>
        <searchType>simple</searchType>
        <searchQuery><xsl:value-of select="search/searchQuery" /></searchQuery>
        <from><xsl:value-of select="search/from" /></from>
        <to><xsl:value-of select="search/to"/></to>
      </search>

  </xsl:template>
```

FIGURA 9.47 - Plantilla XSL utilizada para enriquecer los mensajes entrantes del escenario de búsqueda simple

Este patrón también es utilizado (FIGURA 9.48) para enriquecer los mensajes entrantes con información adicional sobre el usuario que realiza la búsqueda, necesario por el servicio de Auditoría

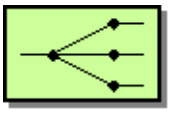
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:bean="http://servicemix.apache.org/bean/1.0"
       xmlns:esb="http://tesis.edu.ar/esb"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/bean/1.0
       http://servicemix.apache.org/schema/servicemix-bean-3.2.2.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <import resource="classpath:clientinfo-bd.xml"/>

  <bean:endpoint service="esb:enricherClientInfo" endpoint="endpoint" bean="#clientInfo"/>

  <bean id="clientInfo" class="ar.edu.tesis.usecase.services.ClientInfoEnricher">
    <property name="dao" ref="clientInfoDAO"/>
  </bean>

</beans>
```

FIGURA 9.48 – Uso del patrón Content Enricher por el servicio de Auditoría

- **Receptor**: el receptor para este flujo de mensajes es un componente que extrae la información que requiere el servicio de búsqueda para realizar la búsqueda en las diferentes fuentes de información registradas. Invoca el servicio de búsqueda y el resultado es retornado al cliente.

C. Servicio basado en el patrón Static Recipient List

Notación

 (imagen extraída de [87])

Implementación en el caso de estudio

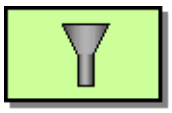
Este patrón es utilizado por los 3 escenarios planteados en el caso de estudio, para realizar invocaciones secuenciales a diferentes servicios. En la FIGURA 9.49 se muestra el extracto de invocaciones secuenciales a 2 servicios desde el router

```
.when(xpath("//searchType='xml'"))
    .to(SEARCH_XML_OUT + "?mep=in-out")
    .to(SEARCH_XSLT_OUT + "?mep=in-out")
```

FIGURA 9.49 - Dos invocaciones secuenciales realizadas a diferentes servicios desde el router

D. Servicio basado en el patrón Message Filter

Notación

 (imagen extraída de [87])

Implementación en el caso de estudio

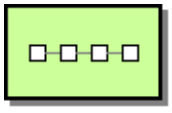
El contenido del mensaje entrante es comparado con un filtro configurado como una expresión XPath (FIGURA 9.50). Si el filtro resulta exitoso entonces el mensaje continua su trayecto, en particular, al servicio de búsqueda. En caso que el filtro no concuerde el mensaje será desechado. Este patrón es utilizado en conjunto con el patrón Content based Router al permitir distinguir entre los 3 escenarios posibles: basados en texto plano, basados en documentos XML o mensajes SOAP.

```
.when(xpath("//searchType='simple'"))
    .to(SEARCH_AUDIT + "?mep=in-out")
    .convertBodyTo(DOMSource.class)
    .to(SEARCH_SIMPLE_OUT + "?mep=in-out")
    .to(SEARCH_AUDIT + "?mep=in-out")
    .to(SEARCH_XSLT_OUT + "?mep=in-out")
```

FIGURA 9.50 – Filtro XPath utilizado en el Router del caso de estudio

E. Servicio basado en el patrón Static Routing Slip

Notación

 (imagen extraída de [87])

Implementación en el caso de estudio

En el caso de estudio este patrón es utilizado para enviar secuencialmente un mensaje a varios destinatarios.

La FIGURA 9.51 es un extracto de la configuración de este servicio para el envío secuencial del mensaje al servicio que implementa el patrón Content Enricher

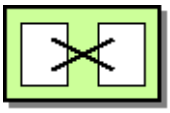
para el escenario de búsqueda simple, y el servicio que implementa el patrón Content based Router

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:eip="http://servicemix.apache.org/eip/1.0"
       xmlns:esb="http://tesis.edu.ar/esb">

  <eip:static-routing-slip service="esb:search-simple-routingSlip" endpoint="endpoint">
        <eip:targets>

          <eip:exchange-target service="esb:search-simple-enricher" />
          <eip:exchange-target service="esb:router" />

       </eip:targets>
 </eip:static-routing-slip>

</beans>
```

FIGURA 9.51 – Configuración del servicio que implementa el patrón Routing Slip

F. Servicio basado en el patrón Message Traslator

Notación

 (imagen extraída de [87])

Implementación en el caso de estudio

En el caso de estudio este patrón es utilizado para transformar los resultados devueltos por el motor de búsqueda al formato requerido por el cliente. En el escenario de búsqueda simple el cliente (humano) requiere que el resultado de la búsqueda sea retornado en formato HTML. El servicio que implementa este patrón se basa en plantillas XSL (FIGURA 9.52) para realizar la transformación.

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="UTF-8" />

<xsl:template match="/">

    <table align="center" width="700px" border="0" cellpadding="1" cellspacing="0">
       <tr>
          <td colspan="4" align="center">
                   <xsl:apply-templates select="searchResponse/searchSummary" />
          </td>
      </tr>
      <tr>
      <td valign="top">
            <div style="position:relative; width:700px; height:300px; overflow: auto;">
             <table align="center" width="700px" border="0" cellpadding="1" cellspacing="0">
                   <xsl:apply-templates select="searchResponse/results" />
             </table>
            </div>
       </td>
      </tr>
    </table>

  </xsl:template>

  <xsl:template match="results">
                     <xsl:apply-templates select="result" />
  </xsl:template>

   <xsl:template match="searchSummary">
          <table class="tablePropiedadesSummary">
             <tr>
```

FIGURA 9.52 - Extracto de la plantilla XSL utilizada para transformar a HTML el resultado de la búsqueda simple

9.9. Componente de Logging

El componente de *logging* permite ir registrando todo lo sucedido dentro de jESBihca. Este componente se basa en la librería Apache Log4J[67], y es configurado en un archivo de configuración XML (FIGURA 9.53) para que todo el proceso de *logging* sea almacenado en una base de datos MySQL (FIGURA 9.54)

```
<appender name="ESB_HIBERNATE" class="ar.edu.tesis.logging.HibernateAppender">

    <param name="threshold" value="DEBUG"/>
    <param name="sessionServiceClass" value="ar.edu.tesis.logging.DefaultHibernateService" />
    <param name="loggingEventClass" value="ar.edu.tesis.logging.DefaultLoggingEvent" />

</appender>
```

FIGURA 9.53 – Extracto configuración log4J

67 **Apache Log4J**: http://logging.apache.org

```
<hibernate-configuration>
    <session-factory>

        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost:3306/tg</property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>

        <property name="connection.pool_size">1</property>

        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <property name="show_sql">true</property>

        <property name="hibernate.hbm2ddl.auto">create</property>

        <mapping resource="LogDocument.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

FIGURA 9.54 – Configuración de los parámetros de conexión a la base de datos de *logging*

Para facilitar el almacenamiento e independizar la base de datos del servicio de *logging*, se utilizó la librería Hibernate, mediante el mapeo declarativo del objeto de Log a una tabla relacional (FIGURA 9.55)

```
<hibernate-mapping>
    <class name="ar.edu.tesis.logging.DefaultLoggingEvent" table="log">
        <id name="id" column="log_id" type="long">
            <generator class="native">
                <param name="sequence">log_id_sequence</param>
            </generator>
        </id>

        <set name="throwableMessages" table="log_throwable" lazy="true">
            <key column="log_id" />
            <composite-element class="ar.edu.tesis.logging.DefaultLoggingEventThrowable">
                <property name="message" column="message" length="255" not-null="true"/>
                <property name="position" column="position" type="integer" not-null="true"/>
            </composite-element>
        </set>
        <property name="level" column="level" length="7" not-null="true" />
        <property name="message" column="message" length="255" not-null="true" />
        <property name="className" column="class_name" length="255" not-null="true" />
        <property name="fileName" column="file_name" length="255" not-null="true" />
        <property name="lineNumber" column="line_number" length="5" not-null="true" />
        <property name="methodName" column="method_name" length="255" not-null="true" />
        <property name="loggerName" column="logger_name" length="255" not-null="true" />
        <property name="startDate" column="start_date" not-null="true" />
        <property name="logDate" column="log_date" not-null="true" />
        <property name="threadName" column="thread_name" length="255" not-null="true" />
    </class>

</hibernate-mapping>
```

FIGURA 9.55 - Extracto del archivo de mapeo objeto/relacional utilizado por Hibernate

9.10. Servicio de Auditoría

Este servicio implementa una auditoría sencilla, la cual es invocada desde el router únicamente para el escenario de búsqueda simple

A. Implementación

Este servicio utiliza Hibernate para hacer persistente la información de auditoría generada por el escenario de búsqueda a partir de un texto simple.

Básicamente el servicio depende de un objeto que implemente la interface IAuditDAO (FIGURA 9.56) la cual es provista por el framework ESB.

Esta implementación es luego inyectada, mediante el framework de Spring, en la implementación del servicio de Auditoría

```
public class AuditDAO extends HibernateDaoSupport implements IAuditDAO {

    public void save(AuditDocument auditDoc) {
        Long searchId = (Long) getHibernateTemplate().save(auditDoc);
    }

    public void update(AuditDocument auditDoc) {
        getHibernateTemplate().update(auditDoc);
    }
```

FIGURA 9.56 - Clase de auditoría que implementa la interface IAuditDAO

B. Configuración

La configuración de este servicio es realizada declarativamente mediante un archivo XML (FIGURA 9.57)

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:bean="http://servicemix.apache.org/bean/1.0"
       xmlns:esb="http://tesis.edu.ar/esb"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/bean/1.0 http://servicemix.apache.org/schema/servicemix-be
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xs


  <import resource="classpath:search-engine-context.xml" />

  <bean:endpoint service="esb:audit" endpoint="auditEndpoint" bean="#auditService"/>

  <bean id="auditService" class="ar.edu.tesis.usecase.services.AuditService">
    <property name="dao" ref="auditDAO"/>
  </bean>

</beans>
```

FIGURA 9.57 – Configuración del servicio de Auditoría

El escenario de búsqueda simple envía como parámetro el nombre del usuario que inicia la búsqueda. A partir de ese usuario el servicio que implementa el patrón Content Enricher (FIGURA 9.58) enriquece el mensaje circulante con información adicional del usuario, recuperada de una base de datos MySQL (FIGURA 9.59).

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:bean="http://servicemix.apache.org/bean/1.0"
       xmlns:esb="http://tesis.edu.ar/esb"
       xmlns:xsi="http://http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://servicemix.apache.org/bean/1.0
       http://servicemix.apache.org/schema/servicemix-bean-3.2.2.xsd
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

  <import resource="classpath:clientinfo-bd.xml"/>

  <bean:endpoint service="esb:enricherClientInfo" endpoint="endpoint" bean="#clientInfo"/>

  <bean id="clientInfo" class="ar.edu.tesis.usecase.services.ClientInfoEnricher">
    <property name="dao" ref="clientInfoDAO"/>
  </bean>

</beans>
```

FIGURA 9.58 – Configuración servicio Content Enricher

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">


    <import resource="classpath:core-data-access-context.xml"/>

    <bean id="clientInfoDAO"
            class="ar.edu.tesis.usecase.services.dao.hibernate.ClientInfoDAOImpl">
        <property name="sessionFactory" ref="hibernateSessionFactory"/>
    </bean>

    <bean id="hibernateSessionFactory" parent="baseSessionFactory">
        <property name="mappingResources">
            <list>
                <value>ar/edu/tesis/usecase/services/ClientInfo.hbm.xml</value>
            </list>
        </property>
    </bean>

</beans>
```

FIGURA 9.59 – Configuración información del usuario utilizada por el servicio de auditoría

La información adicional del usuario es incorporada al mensaje circulante y vuelta a enviar a jESBihca para su enrutamiento al servicio correspondiente (FIGURA 9.60)

```
@Resource
private DeliveryChannel channel;

public void onMessageExchange(MessageExchange exchange) throws MessagingException {

    InOut inOut = (InOut) exchange;

     if (inOut.getStatus() == ExchangeStatus.DONE) {
            return;}
                else if (inOut.getStatus() == ExchangeStatus.ERROR) {return;}

    try {

        NormalizedMessage in = inOut.getInMessage();
        String username = (String) in.getProperty("username");
        ClientInfo ci = dao.getClientInfoByUsername(username);
        in.setProperty("clientInfo", ci);
        inOut.setOutMessage(in);

    } catch (Exception e) {
            e.printStackTrace();
            inOut.setError(e);

    } finally {
            channel.send(inOut);
     }
```

FIGURA 9.60 – Seteo de la información del usuario recuperada de la base de datos

9.11. Servicio de Scheduling

El servicio de scheduling es utilizado para periódicamente invocar otros servicios. En el caso de estudio planteado se utiliza para invocar cada 60 segundos el servicio de indexación (FIGURA 9.61)

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns:quartz="http://servicemix.apache.org/quartz/1.0"
       xmlns:esb="http://tesis.edu.ar/esb">

    <!-- 1.Segundos, 2. Minutos, 3.Horas, 4. Dia del mes, 5.Mes, 6.Dia de la semana, 7.Año(opcional) -->
    <quartz:endpoint service="esb:scheduler" endpoint="endpoint" targetService="esb:indexer">
        <quartz:trigger>
          <quartz:cron cronExpression="0 0/1 * * * ?" />
        </quartz:trigger>
    </quartz:endpoint>

</beans>
```

FIGURA 9.61 - Configuración del servicio de scheduling con la periodicidad con que invoca el servicio destino (esb:indexer)

Capítulo 10

jESBihca: Ejecución escenarios

10.1. Introducción

Este capítulo explica los pasos realizados en la ejecución de cada uno de los 3 escenarios del caso de estudio implementado: escenario de búsqueda basado en texto simple, basado en documentos XML (con formato preacordado) y basado en mensajes SOAP (con formato estándar). La estructura general de ejecución es similar en los tres escenarios.

10.2. Escenarios

A. Cliente Búsqueda a partir de texto simple

A.1. Introducción

La interface que se ofrece al usuario (FIGURA 10.1) es un campo de texto donde deben ingresar el texto, o parte de él, a buscar. El motor de búsqueda implementado, basado en Apache Lucene, soporta distintas variantes en el texto a buscar.

1. Frases encerradas entre comillas dobles, en caso de querer buscar frases completas.

2. Palabras sueltas, eventualmente combinadas con operadores lógicos (AND, OR, NOT)

3. Palabras parciales, y por lo tanto pueden utilizarse caracteres *wildcard* ("?", "*")

En este escenario el formulario de búsqueda puede ser parametrizado mediante los campos:

- **Ordenar Por**: permite ordenar los resultados de acuerdo a su relevancia (default), su tipo de archivo, fuente de datos o autor
- **Res.Desde/Res.Hasta:** permite restringir los resultados a las páginas comenzadas en Res.Desde hasta Res.Hasta
- **Formato Resultado:** este escenario admite 2 posibles formatos en el resultado: HTML (default) o PDF
- **Fuentes de información registradas:** permite filtrar los resultados de las 4 fuentes de información registradas por defecto.

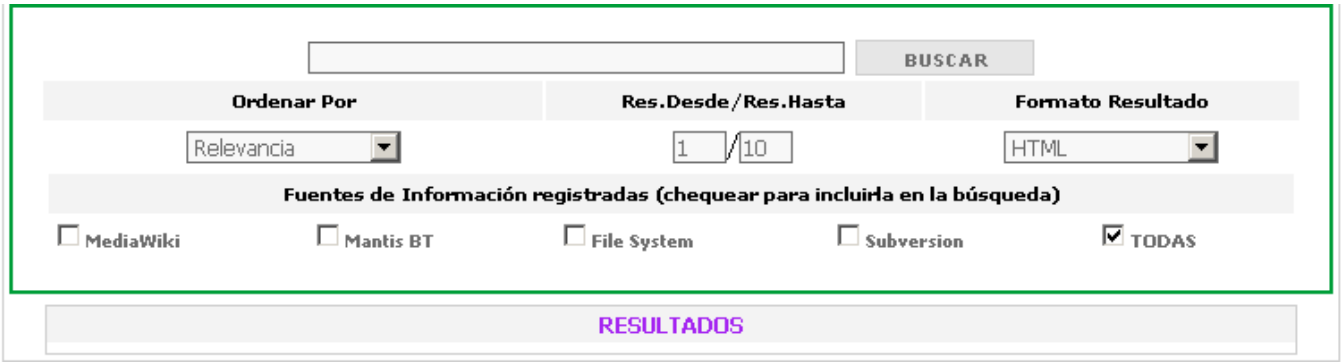


FIGURA 10.1 - Formulario de búsqueda a partir de un texto simple

A.2. Pasos al realizar la búsqueda federalizada

Paso 1

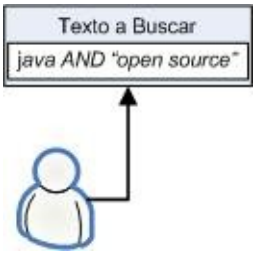


FIGURA 10.2 – Paso 1 escenario de búsqueda simple

El escenario de búsqueda basado en texto simple se inicia cuando el usuario accede a la interface web cliente e ingresa la/s palabra/s a buscar en el campo de texto (es posible parametrizar la búsqueda para "customizar" el resultado) de acuerdo a las distintas alternativas mostradas anteriormente para construir el texto de búsqueda

El submit del formulario envía el requerimiento al servicio HTTP de jESBihca escuchando (por defecto) en el puerto 8192.

Paso 2

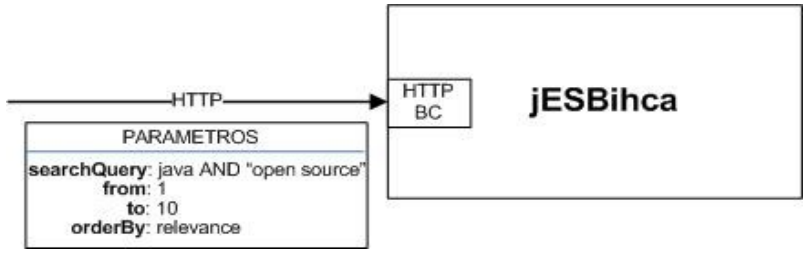


FIGURA 10.3 – Paso 2 escenario de búsqueda simple

El requerimiento, junto a los parámetros de la búsqueda, son enviados por HTTP y atendidos por el servicio HTTP de tipo BC (*Binding Component*) desplegado en JESBihca y escuchando en el puerto 8192 (por defecto).

Paralela e independientemente del procesamiento del requerimiento, JESBihca va realizando el *logging*, (ver Servicio de Logging, en Cap.10, sección 10.8), de los pedidos y respuestas, utilizando la librería Apache Log4J, que lleguen o retorne el ESB

Los logs del flujo de mensajes pueden visualizarse a través del sitio web de administración del caso de estudio.

Paso 3

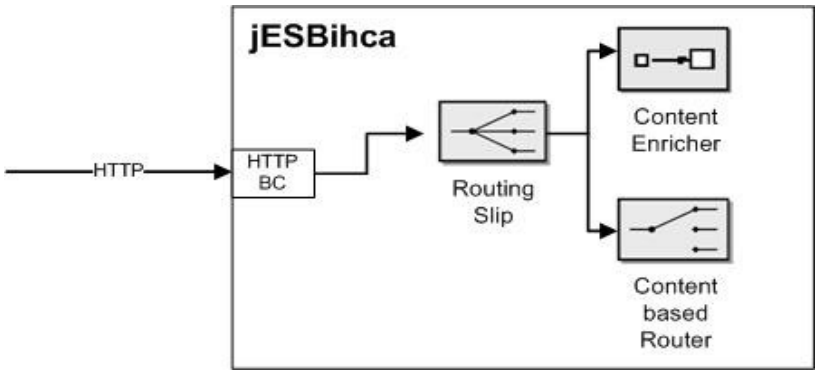


FIGURA 10.4 – Paso 3 del escenario de búsqueda simple

En este paso el requerimiento es atendido secuencialmente por servicios que implementan distintos patrones de integración: Static Routing Slip, Content Enricher y Content Based Router .

Cada uno de estos servicios realiza la lógica de transformación y enrutamiento necesaria para que el mensaje se adecue a los requerimientos pretendidos por el escenario de búsqueda simple.

- El servicio que implementa el patrón de integración *Static Routing Slip* permite redirigir el flujo del mensaje a varios destinatarios, configurados a través de un archivo XML de configuración
- En este escenario este servicio define 2 destinatarios: el servicio que implementa el patrón Content Enricher, con ID *search-simple-enricher*, y el servicio que implementa el patrón Content based Router, con ID *router*. Ambos servicios deberán estar desplegados en el ESB al momento de realizar la invocación.
- El servicio que implementa el patrón de integración Content Enricher, a través de una plantilla XSL, enriquece el contenido del mensaje (en formato XML) agregando información necesaria para el posterior enrutamiento del mismo
- Finalmente el mensaje es enviado al servicio que implementa el patrón de integración Content based Routing (CBR) donde, a partir del tipo de mensaje recibido, *simple, XML o SOAP*, decide la ruta que debe seguir el mensaje. La implementación del router es realizada utilizando la librería Apache Camel.

Paso 4

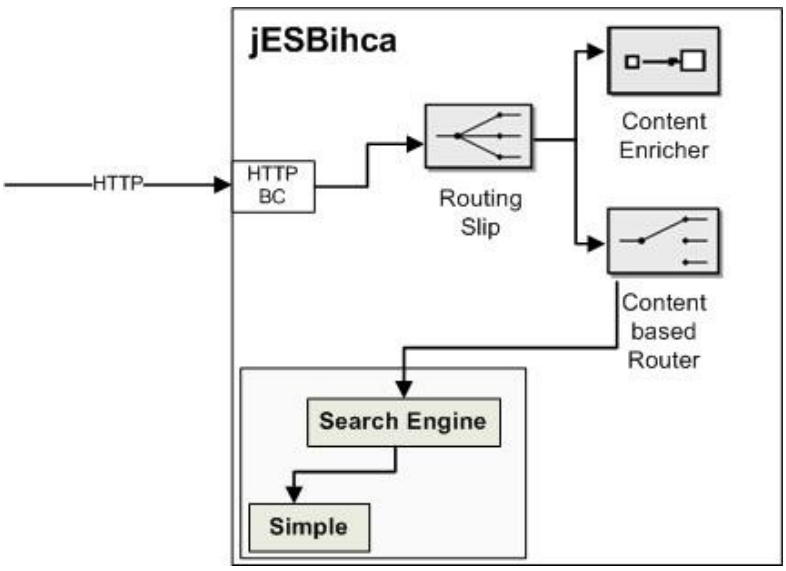


FIGURA 10.5 – Paso 4 del escenario de búsqueda simple

En este paso jESBihca realiza la invocación al servicio de búsqueda simple con los parámetros recibidos desde el usuario.

La implementación del motor de búsqueda basada en Apache Lucene configura el *query* y ejecuta la consulta contra el índice Lucene creado y actualizado por los servicios de indexación y scheduling

Paso 5

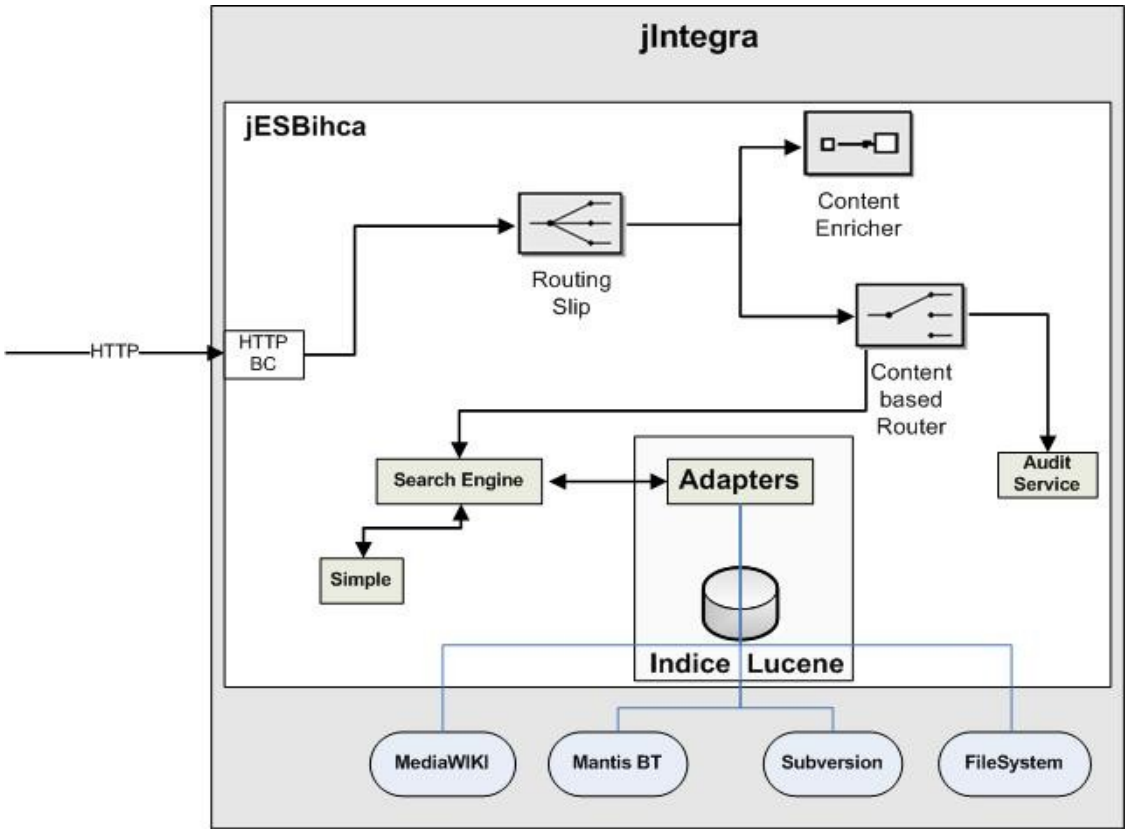


FIGURA 10.6 – Paso 5 del escenario de búsqueda simple

El motor de búsqueda coopera con los adaptadores para realizar la búsqueda federalizada en las distintas fuentes de información (herramientas colaborativas asincrónicas) registradas

Básicamente los adaptadores ofrecen una capa de abstracción a las conexiones específicas necesarias para recuperar información desde las herramientas colaborativas asincrónicas registradas.

Los adaptadores también homogenizan la información extraída de las fuentes de información a un formato único y entendible por la herramienta Apache Lucene.

De esta forma, es posible construir y agregar nuevos adaptadores a distintas herramientas colaborativas, siempre que se respete el formato de documento

Lucene previamente establecido.

Paso 6

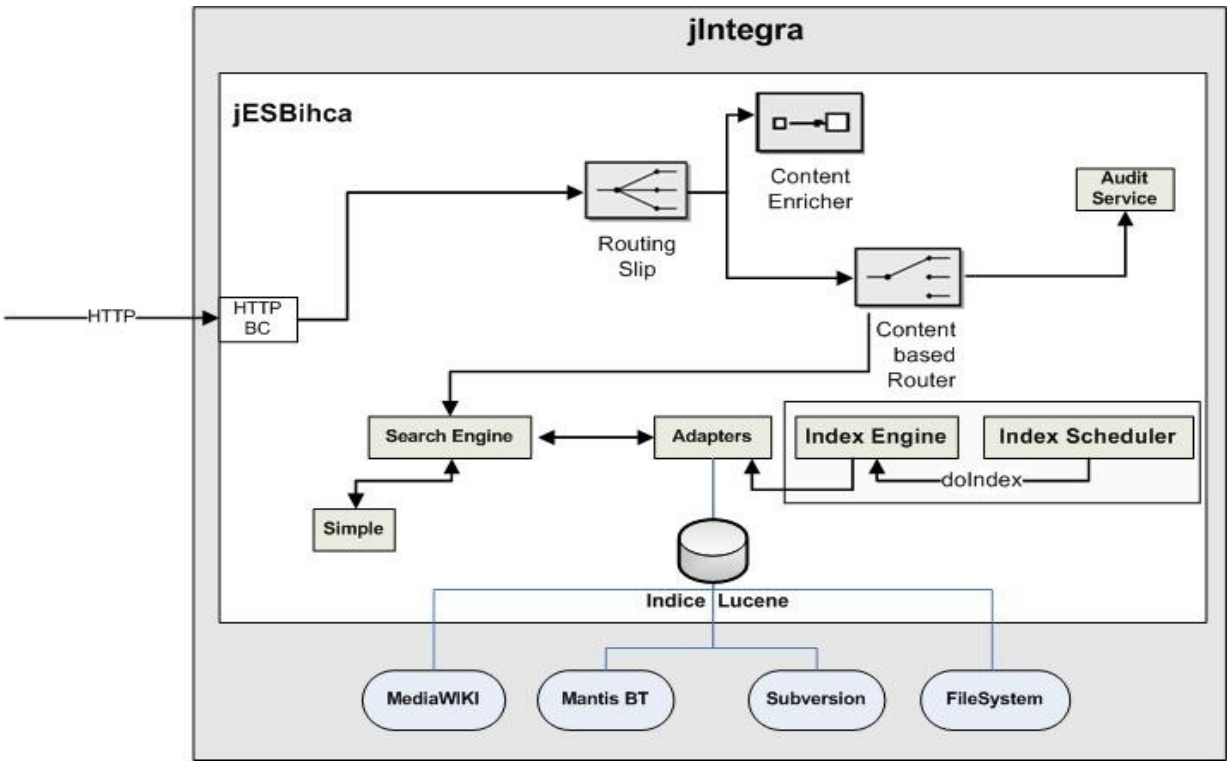


FIGURA 10.7 – Paso 6 del escenario de búsqueda simple

Durante todo el flujo seguido por los mensajes normalizados entre los diferentes servicios desplegados e involucrados en el escenario de búsqueda simple, el servicio de indexación (ver Capítulo 10, sección 10.4.A) indexa y actualiza constantemente el índice Lucene a partir de invocaciones realizadas periódicamente (por defecto cada 60 segundos) por el servicio de scheduling (ver Capítulo 10, sección 10.10)

Paso 7

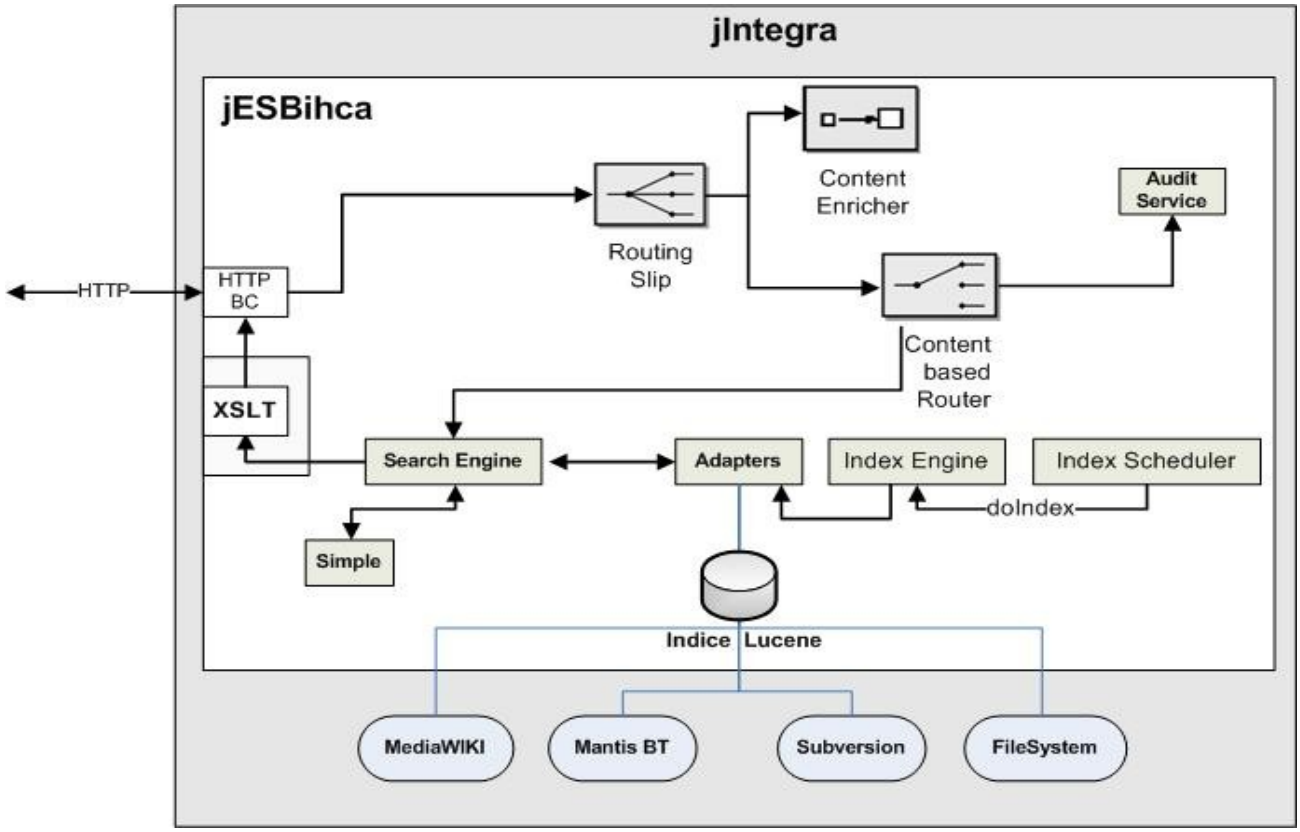


FIGURA 10.8 – Paso 7 del escenario de búsqueda simple

Una vez obtenidos los resultados desde el motor de búsqueda, es necesario adecuar el formato de salida de acuerdo al escenario correcto (búsqueda simple, XML, o SOAP).

En el caso del escenario de búsqueda simple, el formato de salida esperado por el usuario es HTML. Esta transformación es realizada por un servicio XSLT que dado un documento XML con los resultados de la búsqueda lo transforma en un documento HTML utilizando la plantilla XSL adecuada

B. Cliente Búsqueda a partir de documentos XML

B.1. Introducción

En este escenario el servicio XML desplegado en jESBihca espera por documentos XML que contengan el texto, o parte de él, a buscar (FIGURA 10.9)

El motor de búsqueda implementado, basado en Apache Lucene, soporta distintas variantes en el texto a buscar.

1. Frases encerradas entre comillas dobles, en caso de querer buscar frases completas.

2. Palabras sueltas, eventualmente combinadas con operadores lógicos (AND, OR, NOT)

3. Palabras parciales, y por lo tanto pueden utilizarse caracteres wildcard ("?", "*")

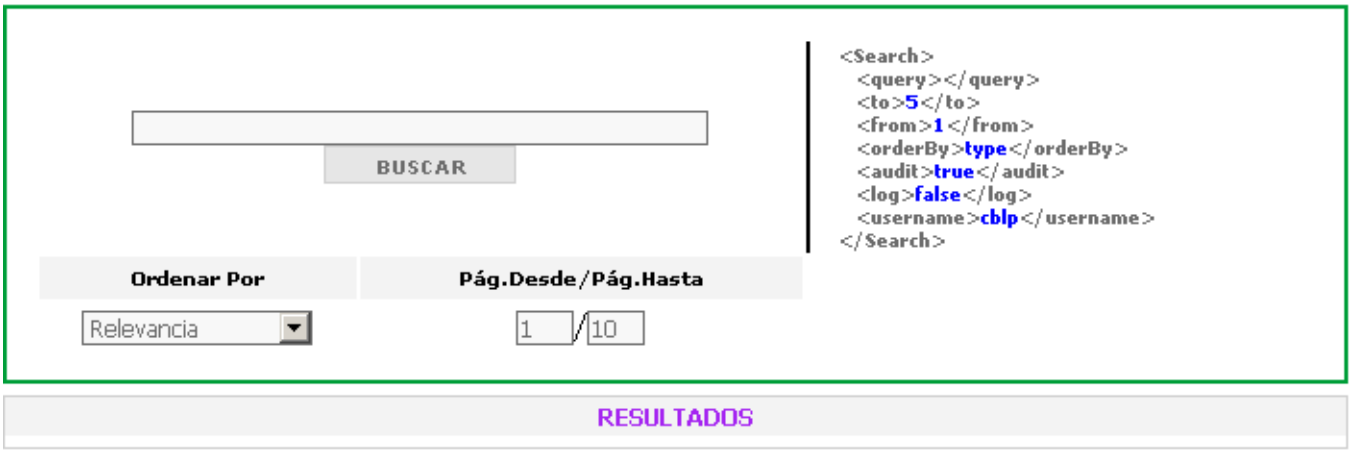


FIGURA 10.9 – Cliente escenario de búsqueda basada en documentos XML

B.2. Pasos al realizar la búsqueda federalizada

Paso 1

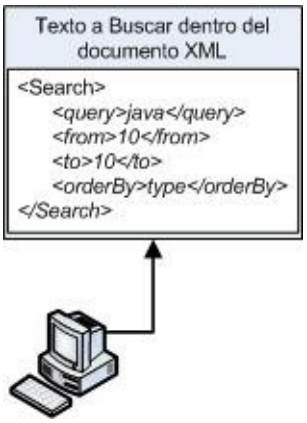


FIGURA 10.10 – Paso 1 del escenario de búsqueda XML

El escenario de búsqueda basado en XML requiere de una aplicación cliente (simulada por el cliente web de la FIGURA 10.9) que construya un documento XML con un formato preestablecido que contenga la/s palabra/s a buscar en el campo de texto, junto a otros parámetros que permitan "customizar" el resultado

El submit del formulario envía el requerimiento al servicio XML de jESBihca desplegado en el puerto 8193

Paso 2

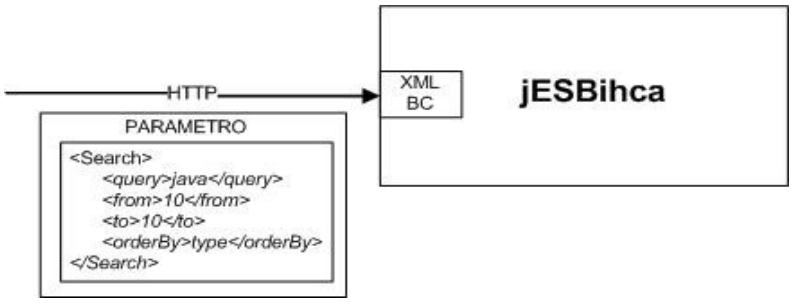


FIGURA 10.11 – Paso 2 del escenario de búsqueda XML

El requerimiento, junto a los parámetros de la búsqueda, son enviados por HTTP y atendidos por el servicio XML desplegado en JESBihca escuchando en el puerto 8193.

Paralela e independientemente del procesamiento del requerimiento, JESBihca va realizando el logueo de los pedidos y respuestas, utilizando la librería Apache Log4J que lleguen o retorne el ESB

Paso 3

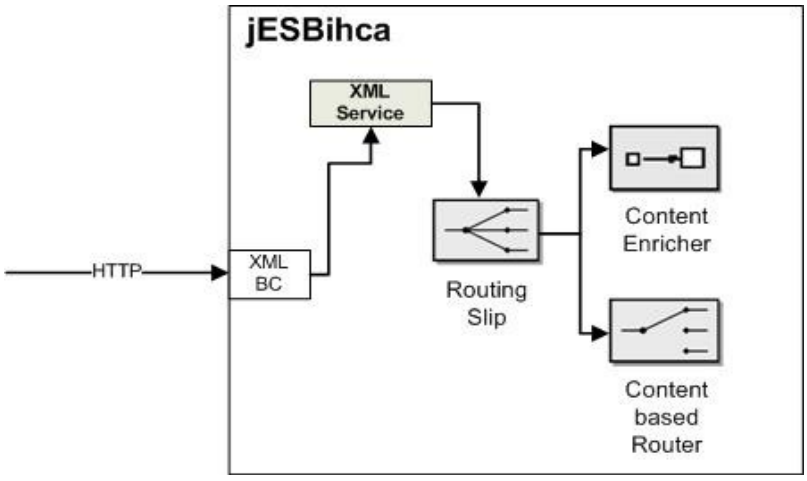


FIGURA 10.12 – Paso 3 del escenario de búsqueda XML

En este paso el requerimiento es atendido secuencialmente por servicios que implementan distintos patrones de integración: Static Routing Slip, Content Enricher y Content based router

Cada uno de estos servicios realiza la lógica de transformación y enrutamiento necesaria para que el mensaje se adecue a los requerimientos pretendidos por el caso de uso XML.

- El servicio que implementa el patrón de integración Static Routing Slip permite redirigir el flujo del mensaje a varios destinatarios, configurados a través de un archivo XML de configuración externo.
- En este escenario este servicio define 2 destinatarios: el servicio que implementa el patrón Content Enricher, con ID *enricherSearchXMLXslt*, y el servicio que implementa el patrón Content based Router, con ID *router*. Ambos servicios deberán estar desplegados en el ESB al momento de realizar la invocación.
- El servicio que implementa el patrón de integración Content Enricher, a través de una plantilla XSL, enriquece el contenido del mensaje (en formato XML) agregando información necesaria para el posterior enrutamiento del mismo
- Finalmente el mensaje es enviado al servicio que implementa el patrón de integración Content based Routing (CBR) donde, a partir del tipo de mensaje recibido: *simple, XML o SOAP*, decide la ruta que debe seguir. La implementación del router es realizada utilizando la librería Apache Camel.

Paso 4

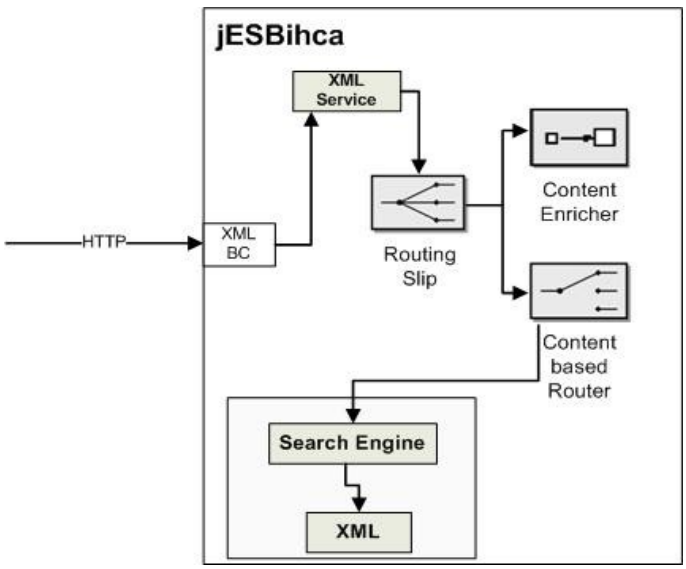


FIGURA 10.13 – Paso 4 del escenario de búsqueda XML

En este paso jESBihca realiza la invocación al servicio de búsqueda xml con documento XML recibido desde la aplicación cliente.

La implementación del motor de búsqueda basada en Apache Lucene configura el query y ejecuta la consulta contra el índice Lucene creado y actualizado por los servicios de indexación y scheduling

Paso 5

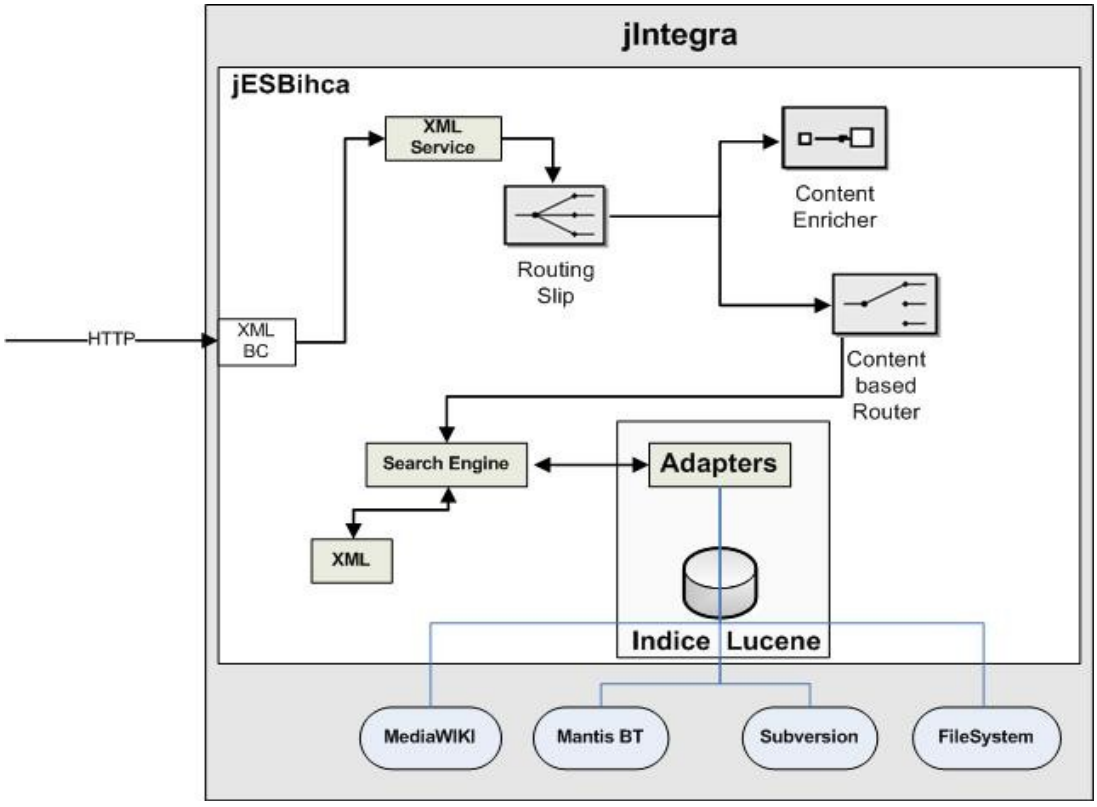


FIGURA 10.14 – Paso 5 del escenario de búsqueda XML

El motor de búsqueda coopera con los adaptadores para realizar la búsqueda federalizada en las distintas fuentes de información (herramientas colaborativas) registradas .

Los adaptadores están definidos declarativamente en un archivo de configuración XML, el cual hereda funcionalidad provista por el framework jESBihca a partir de archivos XMLs

Paso 6

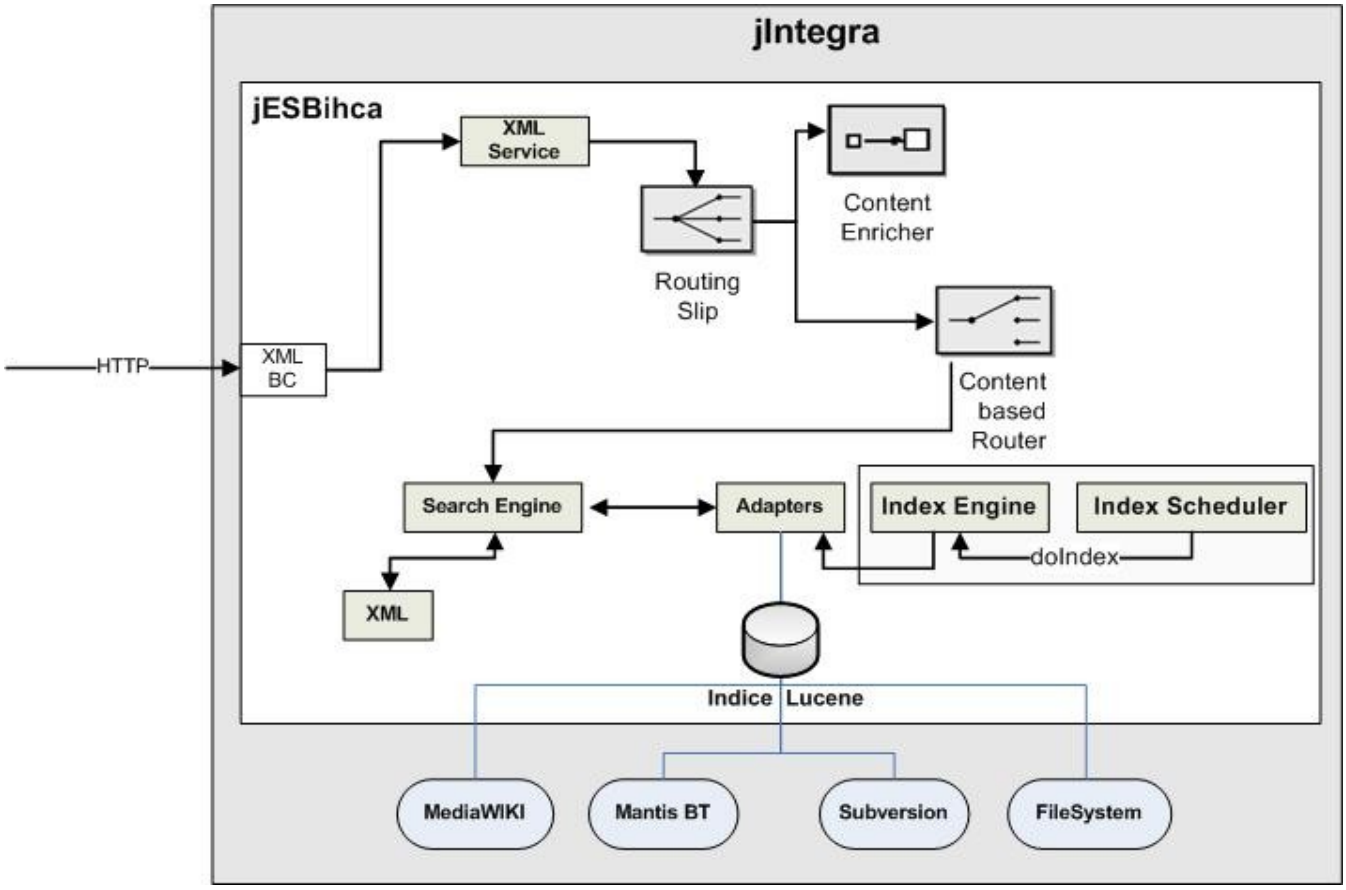


FIGURA 10.15 – Paso 6 del escenario de búsqueda XML

Durante todo el flujo seguido por los mensajes normalizados entre los diferentes servicios desplegados e involucrados en el escenario de búsqueda XML, el servicio de indexación (ver Capítulo 10, sección 10.4.A) indexa y actualiza constantemente el índice Lucene a partir de invocaciones realizadas periódicamente (por defecto cada 60 segundos) por el servicio de scheduling (ver Capítulo 10, sección 10.10)

Paso 7

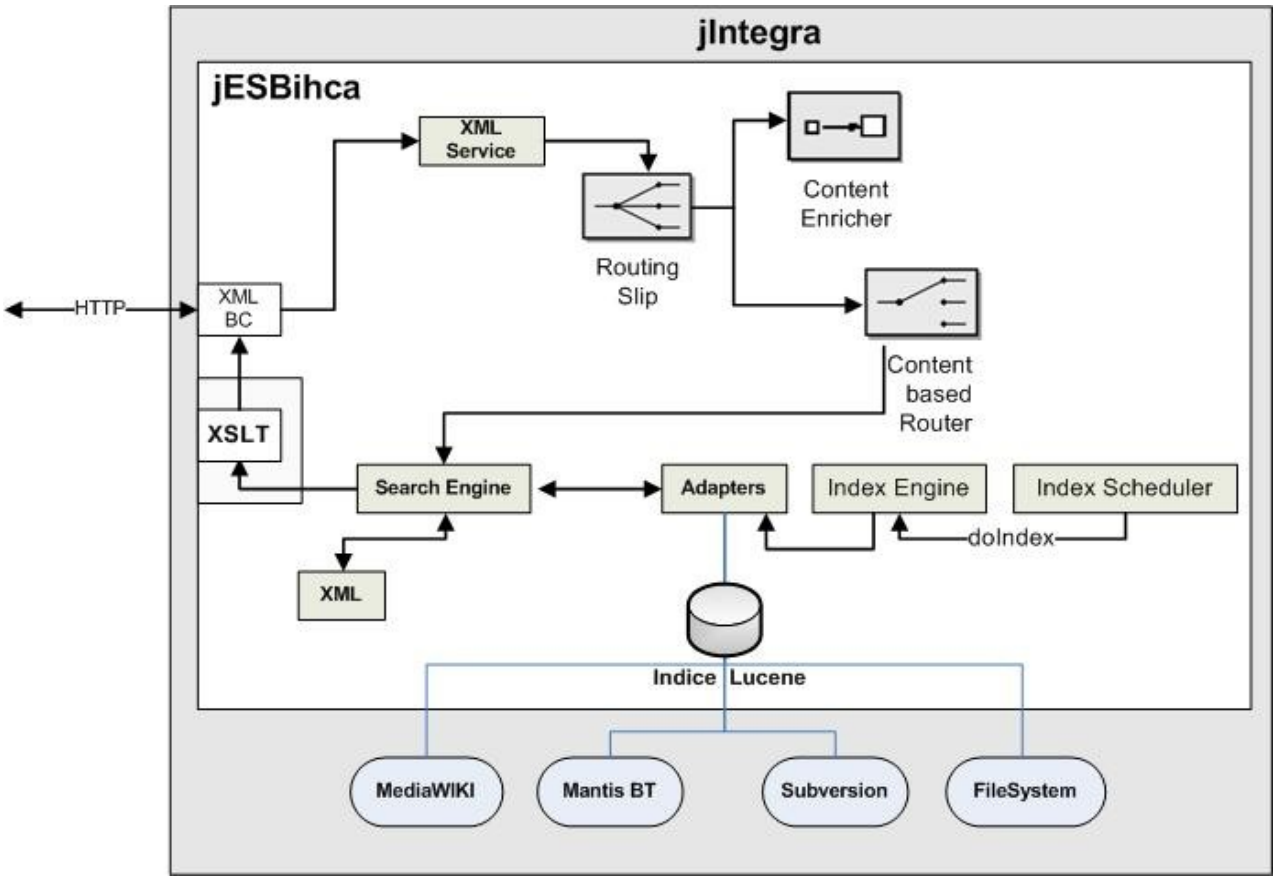


FIGURA 10.16 – Paso 7 del escenario de búsqueda XML

Una vez obtenidos los resultados desde el motor de búsqueda, es necesario adecuar el formato de salida de acuerdo al escenario correcto (búsqueda simple, XML, o SOAP).

En el caso del escenario de búsqueda XML, el formato de salida esperado por la aplicación cliente es XML.

Sin embargo, por cuestiones de practicidad y para que las pruebas sobre este escenario sean fácilmente realizables, este escenario también devuelve el resultado en formato HTML, al invocarse desde el servicio Router al servicio de transformación basado en plantillas XSL

C. Cliente Búsqueda a partir de mensajes SOAP

C.1. Introducción

En este escenario el servicio SOAP desplegado en jESBihca espera por mensajes SOAP que contengan el texto, o parte de él, a buscar (FIGURA 10.17)

El motor de búsqueda implementado, basado en Apache Lucene, soporta distintas variantes en el texto a buscar.

1. Frases encerradas entre comillas dobles, en caso de querer buscar frases completas.

2. Palabras sueltas, eventualmente combinadas con operadores lógicos (AND, OR, NOT)

3. Palabras parciales, y por lo tanto pueden utilizarse caracteres *wildcard* ("?", "*")

FIGURA 10.17 – Cliente web para el servicio de búsqueda basado en mensajes SOAP

C.2. Pasos

Paso 1

FIGURA 10.18 – Paso 1 del escenario de búsqueda SOAP

El escenario de búsqueda basado en mensajes SOAP requiere de una aplicación cliente (simulada por el cliente web de la FIGURA 10.17) que construya un mensaje SOAP que contenga la/s palabra/s a buscar en el campo de texto, junto a otros parámetros que permitan "customizar" el resultado

El submit del formulario envía el requerimiento al servicio SOAP de jESBihca desplegado en el puerto 8194.

Paso 2

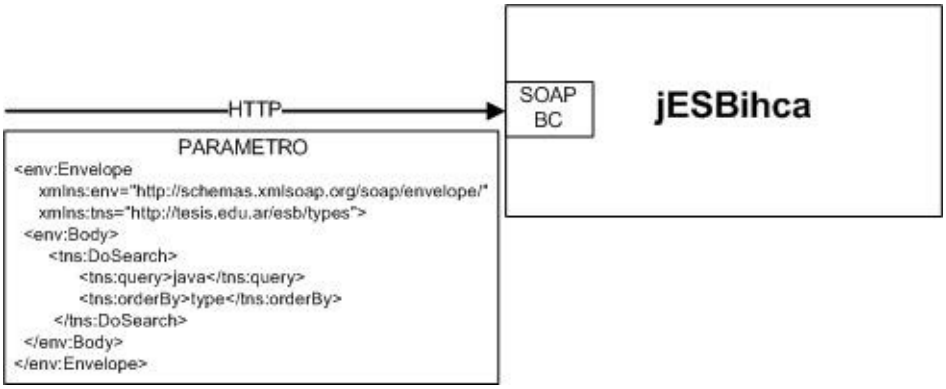


FIGURA 10.19 – Paso 2 del escenario de búsqueda SOAP

El requerimiento, junto a los parámetros de la búsqueda, son enviados por HTTP y atendidos por el servicio SOAP desplegado en JESBihca escuchando en el puerto 8194.

Paralela e independientemente del procesamiento del requerimiento, JESBihca va realizando el *logging* de los pedidos y respuestas (ver Capítulo 10, sección 10.8), que lleguen o retorne el ESB

Los logs del flujo de mensajes pueden visualizarse a través del sitio web de administración

Paso 3

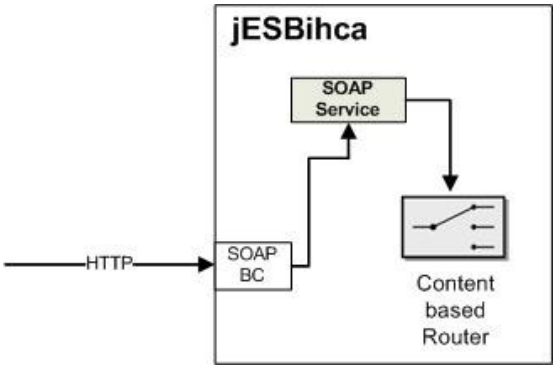


FIGURA 10.20 – Paso 3 del escenario de búsqueda SOAP

En este paso el requerimiento es atendido por el servicio que implementa el patrón de integración Content based router

Este servicio realiza la lógica de enrutamiento necesaria para que el mensaje SOAP invoque los servicios necesarios para adecuarse a los requerimientos pretendidos por este escenario.

El mensaje SOAP es enviado al servicio que implementa el patrón de integración Content based Routing (CBR) donde, a partir del tipo de mensaje recibido: *simple, XML o SOAP*, decide la ruta que debe seguir. La implementación del router es realizada utilizando la librería Apache Camel.

Paso 4

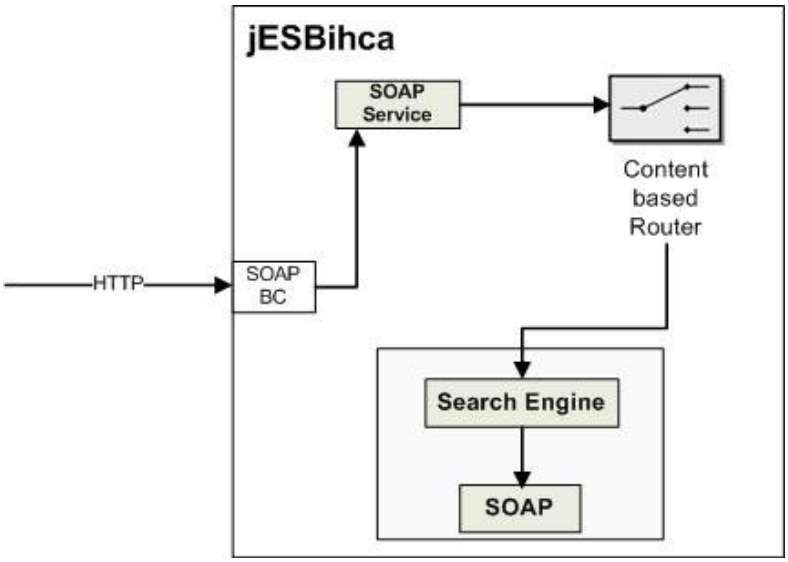


FIGURA 10.21 – Paso 4 del escenario de búsqueda SOAP

En este paso jESBihca realiza la invocación al servicio de búsqueda SOAP con el mensaje SOAP recibido desde el cliente.

La implementación del motor de búsqueda basada en Apache Lucene configura el query y ejecuta la consulta contra el índice Lucene creado y actualizado por los servicios de indexación y scheduling.

Paso 5

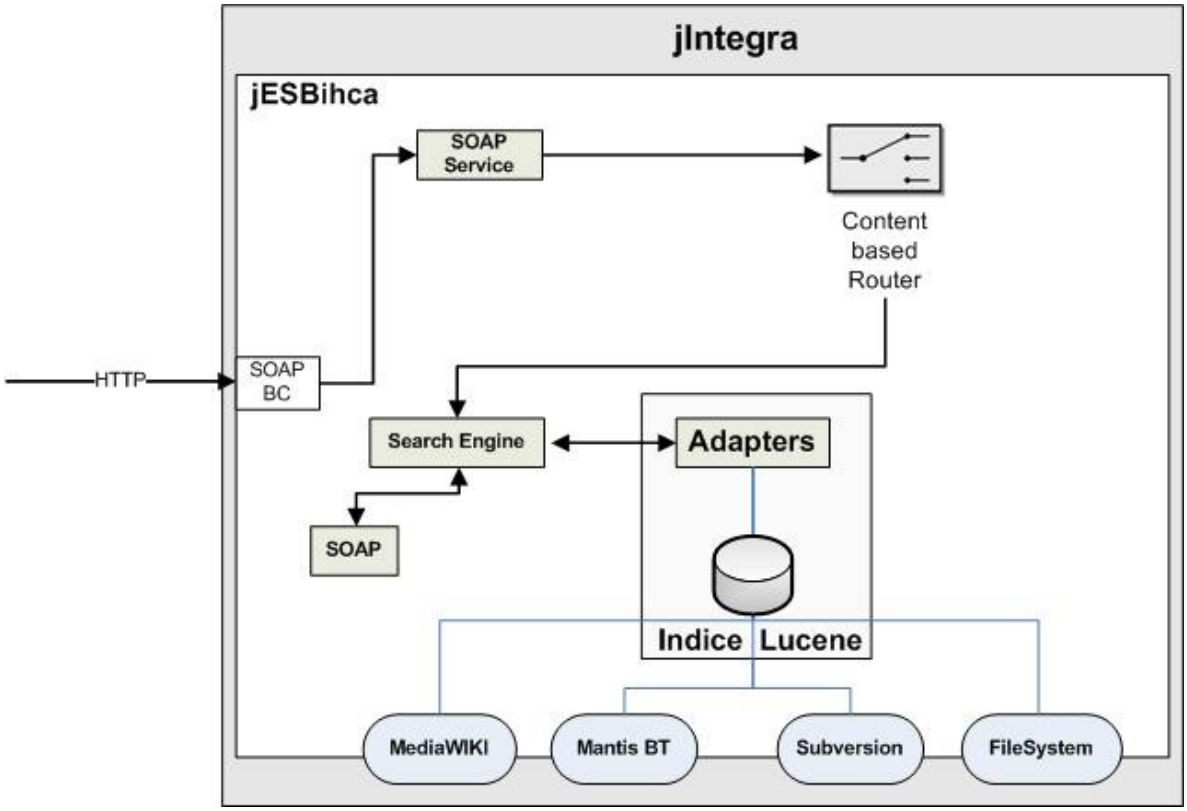


FIGURA 10.22 – Paso 5 del escenario de búsqueda SOAP

El motor de búsqueda coopera con los adaptadores para realizar la búsqueda federalizada en las distintas fuentes de información (herramientas colaborativas) registradas.

Los adaptadores están definidos declarativamente en un archivo de configuración XML, el cual hereda funcionalidad provista por el framework jESBihca a partir de archivos XMLs.

Paso 6

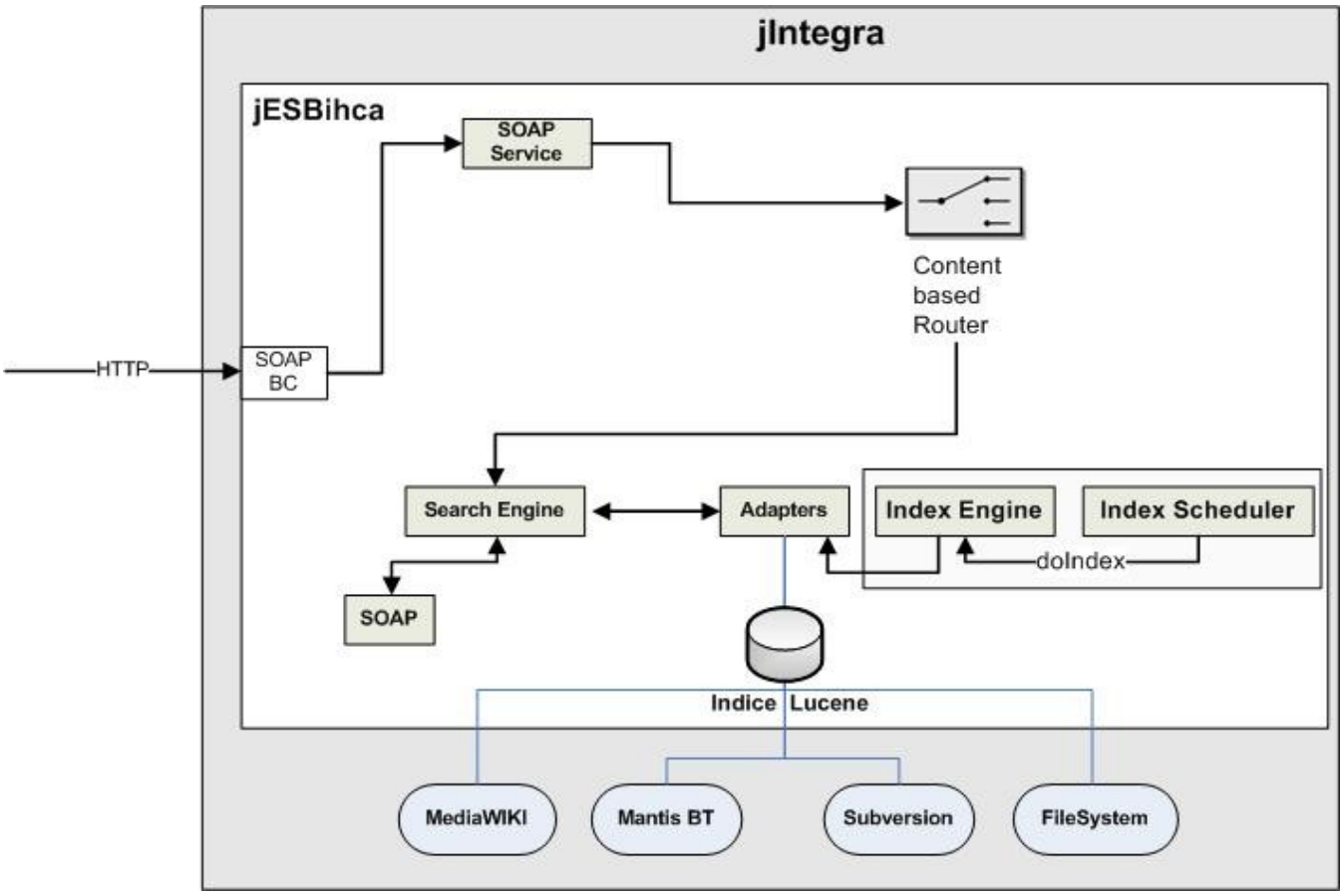


FIGURA 10.23 – Paso 6 del escenario de búsqueda SOAP

Durante todo el flujo seguido por los mensajes normalizados entre los diferentes servicios desplegados e involucrados en el escenario de búsqueda SOAP, el servicio de indexación (ver Capítulo 10, sección 10.4.A) indexa y actualiza constantemente el índice Lucene a partir de invocaciones realizadas periódicamente (por defecto cada 60 segundos) por el servicio de scheduling (ver Capítulo 10, sección 10.10)

Paso 7

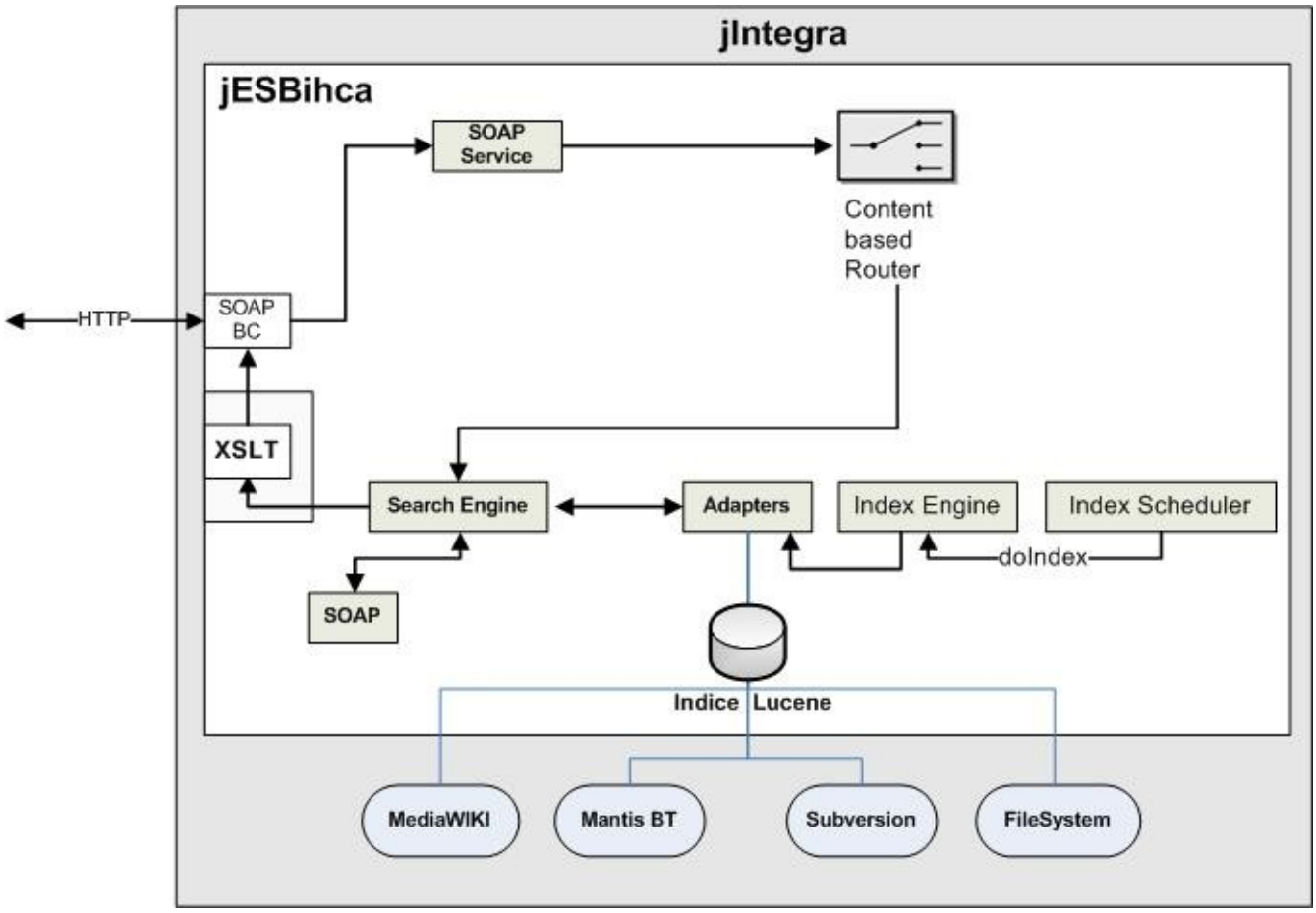


FIGURA 10.24 – Paso 7 del escenario de búsqueda SOAP

Una vez obtenidos los resultados desde el motor de búsqueda, es necesario adecuar el formato de salida de acuerdo al escenario correcto (búsqueda simple, XML, o SOAP).

En el caso del escenario de búsqueda SOAP, el formato de salida esperado por el usuario es XML (con un formato estándar).

Sin embargo, por cuestiones de practicidad y para que las pruebas sobre este escenario sean fácilmente realizables, este caso de uso también devuelve el resultado en formato HTML, al invocarse en última instancia, y desde el servicio ruteador, al servicio de transformación basado en plantillas XSL

Anexo I

Orientación a Servicios y su relación con la Orientación a Objetos

1. Introducción

En el capítulo 3, sección 3.1.1, se realizó una introducción a conceptos relacionados con la orientación a servicios, que facilita el entendimiento y objetivos de las arquitecturas orientadas a servicios.

Este anexo pretende complementar la introducción de la sección 3.1.1, ofreciendo un análisis no exhaustivo sobre la relación entre la orientación a servicios y un paradigma ya establecido, y bien conocido como es el diseño orientado a objetos. Es necesario remarcar que ambos paradigmas no son excluyentes, sino más bien complementarios, y desde esta visión se realizará el análisis.

Además, como explica Thomas Erl[68], de no haber sido por los principios y patrones de diseño innovativos y formalizados por la orientación a objetos, seguramente el modelo de arquitecturas orientadas a servicios y el framework de servicios web no hubiesen existidos tal cual lo conocemos actualmente [13]

68 **Thomas Erl**: http://www.thomaserl.com

2. Relación entre los paradigmas

El diseño orientado a objetos popularizó la visión de construir aplicaciones a partir de elementos de software reusables y flexibles.

Surge de la necesidad de ordenar y organizar los procesos de desarrollo no estructurados que generaban varios problemas, especialmente el conocido como “código spaghetti”.

Este paradigma combinó buenas prácticas de la programación estructurada con sólidos principios de diseño, que permitieron la construcción de software a partir de unidades que reflejaban cercanamente los objetos del mundo real.

La orientación a objetos pretende maximizar la vida útil de las aplicaciones en base a los requerimientos de negocios, incluyendo actualizaciones posteriores y extensiones. A partir de reglas y guías que gobiernan cuidadosamente la separación de la lógica de aplicación y datos en objetos que pueden ser individualmente mantenidos, se pretende minimizar el impacto que puede causar a las aplicaciones los cambios del ambiente.

Por su parte la orientación a servicios comparte muchos de los objetivos del diseño orientado a objetos. Busca establecer principios de diseño flexibles que puedan adaptarse a los constantes cambios en los requerimientos de negocios en las organizaciones. Similar al diseño orientado a objetos, la orientación a servicios pretende minimizar el impacto del cambio sobre las aplicaciones que ya están en uso.

Los principios que promueven el débil acoplamiento y la composición de servicios colaboran en la evolución de los servicios implementados conjuntamente con los cambios de requerimientos de negocio.

Una distinción común entre ambos paradigmas es en cuanto a su alcance. Aunque la orientación a objetos no limita explícitamente el alcance en que son aplicados sus principios, usualmente son usados en aplicaciones aisladas, o conjunto de aplicaciones relacionadas. En cuanto a reusabilidad, generalmente se implementa a partir del uso de componentes compartidos o comunes, reusados por diferentes aplicaciones.

Históricamente el diseño orientado a objetos ha sido aplicado a segmentos específicos dentro de la organización, mientras que la orientación a servicios pretende abarcar la mayor parte de, o toda, la organización (FIGURA A.1)

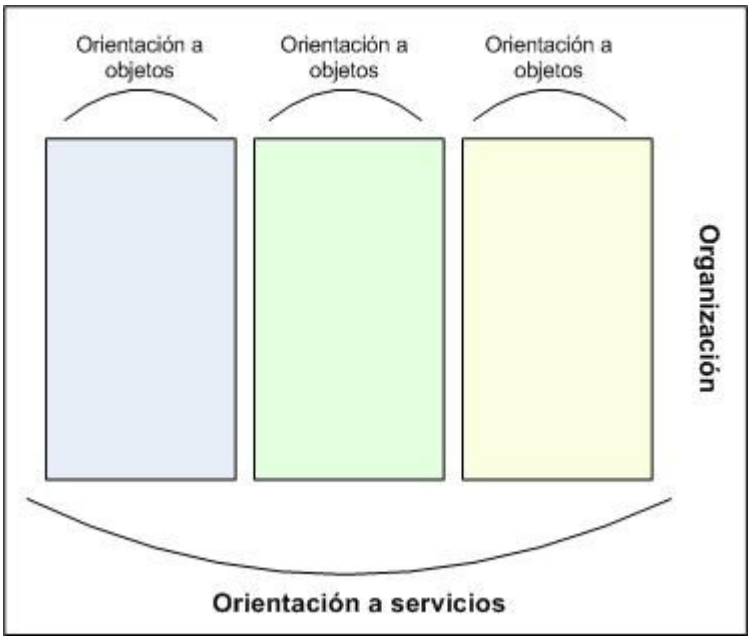


FIGURA A.1 – Relación Orientación a Objetos y Orientación a Servicios

Es necesario remarcar, además, que algunos principios y patrones del diseño orientado a objetos surgen en tiempos en que la mayoría de las organizaciones construían aplicaciones basadas en componentes, distribuídas, usando tecnología RPC. El reuso de cualquier objeto estaba enmarcado en los límites propios de la plataforma RPC.

Las limitaciones propias de RPC para conectarse con otras plataformas tecnológicas facilitaron el surgimiento de EAI (*Enterprise Application Integration*), fuerte influencia de la orientación a servicios en general, y en particular de algunas de las tecnologías analizadas en esta tesis.

A pesar que la orientación a objetos tuvo fuerte influencia en SOA y la orientación a servicios, su consolidación fue gracias al surgimiento del framework de servicios web. Si bien al principio la funcionalidad ofrecida por los servicios web era más bien primitiva, permitió establecer el potencial para romper con las limitaciones de las plataformas y aplicaciones propietarias, como así también inspirar la visión de la interconectividad y federación entre organizaciones.

El modelo arquitectónico subyacente de SOA y los principios detrás de la orientación a servicios fueron desarrollados en base a tal visión. Como resultado se logra gran sinergia con la segunda generación de servicios web, otras tecnologías y estándares. La FIGURA A.2 ilustra las influencias de la orientación a servicios.

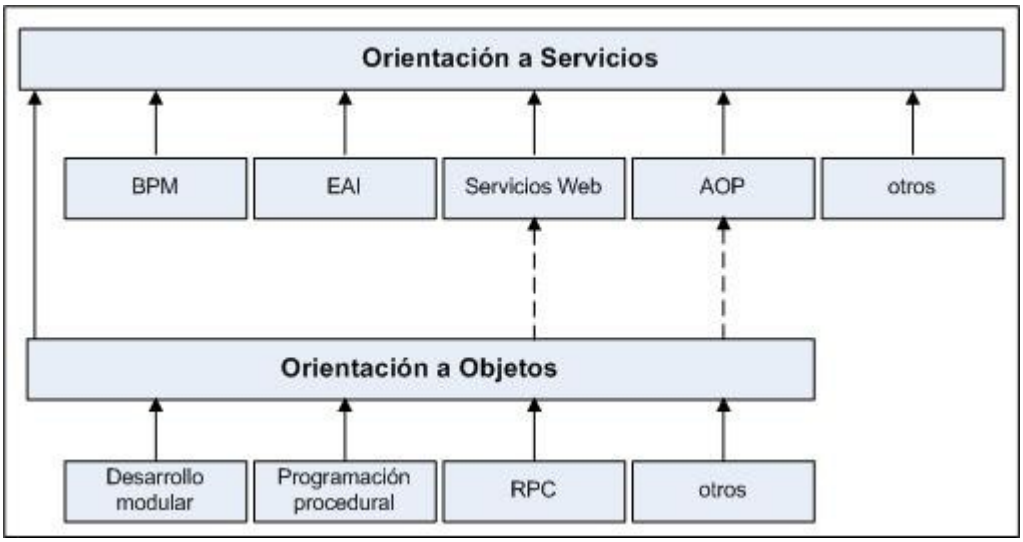


FIGURA A.2 – Influencias orientación a servicios

3. Comparación de conceptos

La orientación a objetos provee clases como el medio para organizar la lógica de la aplicación, que funcionan como contenedores de comportamiento y datos. De esta manera, una clase puede ser vista como una plantilla de la cual se crean instancias en ejecución, denominadas objetos, cada cual con su estado y comportamiento propio. El comportamiento y estado de las clases son definidos a partir de métodos y atributos, respectivamente.

3.1. Mensajes

La comunicación entre objetos se realiza mediante el intercambio de mensajes. El nombre mensaje es utilizado genéricamente en el diseño orientado a objetos, pero no tiene relación alguna en cuanto a detalles de su implementación física. En el caso de componentes que se comuniquen con protocolos de comunicación no estándares (usualmente basados en RPC), los mensajes son expresados como unidades binarias de comunicación intercambiadas sincrónicamente.

Los mensajes usados por servicios implementados como servicios web, típicamente se corresponden a unidades basadas en texto que pueden ser intercambiadas sincrónica o asincrónicamente.

Los métodos de objetos son generalmente diseñados para ***portType*** intercambiar datos de granularidad fina, ya que la conexión que establecen con los demás objetos, ya sean locales o remotos, es usualmente persistente.

En cambio, los servicios web típicamente utilizan el protocolo sin estado HTTP, para el intercambio de mensajes. Debido a no tener el beneficio de una conexión

persistente (o con estado), las operaciones deben ser diseñadas para el intercambio de mensajes que contengan toda la información requerida por el servicio destinatario.

3.2. Interfaces

Una colección de métodos pueden ser definidos, no así implementados, dentro de una interface. Una clase puede implementar una interface estableciendo, de esta manera, un *endpoint* formal de la lógica encapsulada en la clase.

La orientación a servicios se focaliza tanto en la definición del contrato de servicio, como en la lógica que éste encapsula. Un contrato de servicio es comparable a la clase que implementa la interface, al ofrecer un punto de entrada a la funcionalidad provista por el servicio, mientras oculta detalles específicos de la implementación.

En el ámbito de servicios web, los documentos WSDL son utilizados para la definición formal de los contratos de servicios. El elemento *portType* de WSDL establece las operaciones del servicio web. De esta manera, este elemento se asemeja[69] al concepto de *interface* de la orientación a objetos. Un servicio web puede declarar múltiples construcciones *portType*, similar a la posibilidad que tienen las clases de implementar múltiples *interfaces.*

69 En la versión 2.0 de WSDL, el elemento ***portType*** se renombra a ***interface***

Anexo II

Conclusiones

La construcción de una arquitectura de integración orientada a servicios raramente implica la creación "desde cero" de los servicios involucrados en la integración. Usualmente aplicaciones existentes sirven como la base para la construcción de tales servicios que formarán parte de SOA.

La tecnología ESB provee la infraestructura de software de mediación que facilita la interconexión entre estos servicios. Externalizar la lógica de mediación fuera de los servicios remueve el acoplamiento entre los servicios conectados al bus. De esta manera la tecnología ESB promueve la integración orientada a servicios en la forma de mediación, que facilita la comunicación entre servicios.

El haber considerado el uso y adaptación de FLOSS para la implementación de *jESBihca* era parte del desafío propuesto por esta tesis, especialmente por:

- Mi fuerte convicción sobre las ventajas tecnológicas y sociales que FLOSS tiene respecto a su contraparte, el software privativo (que limita las libertades de los usuarios, y cuyo código fuente está oculto).
- Ser conciente sobre los limitados conocimientos, y recursos técnicos que tenía para poder realizar la implementación de una solución de integración que cumpliera todos los aspectos y capacidades mínimas requeridas por un ESB, y principalmente que se adaptara a algún estándar adoptado por la industria del software. No hubiese sido una tarea fácil para mí, e incluso siquiera posible, el analizar, diseñar e implementar un ESB "desde cero" que permita la búsqueda y recuperación de información dispersa en diferentes herramientas colaborativas y asincrónicas, sin reutilizar herramientas de desarrollo FLOSS.

- Poder contribuir con la sociedad a partir de la implementación de una solución basada íntegramente en herramientas con licencias FLOSS, y que ayude a resolver una problemática particular.

Es así que las herramientas de desarrollo y proyectos sobre los cuales se construyó *jESBihca* tuvieron su preferencia en aquellas que tuviesen licencias FLOSS, que me permitieran estudiar, modificar, adaptar, reutilizar y optimizar el funcionamiento originalmente provisto, gracias a la disponibilidad del código fuente, y que no tuvieran costo de licencia para su uso, adaptación, optimización y redistribución.

JESBihca es, pues, el resultado tangible de los temas teóricos presentados y analizados en los capítulos que comprenden la parte 1 de este informe, siendo su utilidad práctica de diversa índole. Se destaca, especialmente, su posible uso en la recuperación y homogenización de información almacenada en diversas y heterogéneas fuentes de datos, para su eventual clasificación y posterior análisis estadístico

Anexo III

Abreviaturas

BC	Binding Component	**BPEL**	Business Process Execution Language
CMS	Content Management Systems	**CSV**	Comma-separated Values
EAI	Enterprise Application Integration	**EDI**	Electronic Data Interchange
ESB	Enterprise Service Bus	**FLOSS**	Free/Libre Open Source Software
JBI	Java Business Integration	**JDK**	Java Development Kit
JEE	Java Enterprise Edition	**JMS**	Java Message Service
JMX	Java Management eXtensions	**MEP**	Message Exchange Pattern
MOM	Message-oriented Middleware	**ORB**	Object Request Broker
ORM	Object-Relational Mapping	**RMI**	Remote Method Invocation

RPC	Remote Procedure Call	**SCM**	Software Control Management
SE	Service Engine	**SGML**	Standard Generalized Markup Language
SNMP	Simple Network Management Protocol	**SOA**	Service-oriented Architecture
SOAP	Simple Object Access Protocol	**SOI**	Service-oriented Integration
UDDI	Universal Description, Discovery and Integration		
XML	Extensible Markup Language	**XSL**	Extensible Stylesheet Language
WSDL	Web Services Description Language		

Anexo IV

Referencias

[1] Stallmann, R. M. and J. Gay: *Free Software, Free Society: Selected Esseys of Richard M. Stallman*. GNU Press. 2002

[2] Eric Raymond: *The Cathedral & the Bazaar*. 2001

[3] Eric Von Hippel: *Democratizing Innovation*. MIT Press, 2005

[4] Kimble, C., P. M. Hildreth: *Knowledge networks: innovation through communities of practice*. Idea Group Publishing,2004

[5] Optaros white paper: *Understanding Free and Open Source Licenses. Version 2.1*. Online en: http://www.optaros.com/news/white-papers-reports

[6] Sveiby, K.: *Tacit Knowledge*. http://www.sveiby.com/articles/Polanyi.html. 1997

[7] Tuomi, I.: *Internet, Innovation, and Open Source:Actors in the Network*. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/824/733 - First Monday, Vol. 6, Number 1, 2001

[8] Gregor Hohpe, Bobby Woolf: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison Wesley, 2004

[9] Matjaz B. Juric, Ramesh Loganathan, Poornachandra Sarang,Frank Jennings: *SOA Approach to integration*. Pack Publishing, 2007

[10] George Coulouris: *Sistemas Distribuidos*. Tercera Edición. Addison Wesley, 2001.

[11] Dirk Krafzig: *Enterprise SOA: Service Oriented Architecture Best Practices*. Prentice Hall, 2004

[12] Eric Pulier, Hugh Taylor: *Understanding Enterprise SOA*. Manning, 2005

[13] Thomas Erl: *SOA Principles of Service Design*. Prentice Hall, 2007

[14] David S. Linthicum: *Next Generation Application Integration: From Simple Information to Web Services.* Addison Wesley, 2003

[15] Nicolai M. Josuttis: *SOA in Practice*. O'Reilly, 2007

[16] Binildas A.Christudas, Malhar Barai, Vincenzo Caselli: *Service-oriented Architectures with Java*. Pack Publishing, 2008

[17] Thomas Erl: *Service-oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005

[18] Thomas Erl: *Service Oriented Architecture: A Field Guide to Integrating XML and Web Services.* Prentice Hall, 2004

[19] Means Scott: *XML in a Nutshell: A Desktop Quick Reference*. O'Reilly, 2001

[20] *Java Technology and XML.* Sun Microsystems. 2002; Online en: http://java.sun.com/xml

[21] IBM Web Services Architecture Team. *Web Services architecture overview*. IBM developerWorks, 2000

[22] SOAP Especificación: http://www.w3.org/TR/soap

[23] WSDL Especificación: http://www.w3.org/TR/wsdl

[24] Eric Newcomer, Greg Lomow: *Understanding SOA with Web Services*. Addison Wesley, 2004

[25] Tijs Rademakers, Jos Dirken: *Open Source ESBs in Action, Capítulo 1*. Manning, 2007

[26] Michael Czapski; Sebastian Krueger; Brendan Marry; Saurabh Sahai; Peter Vaneris; Andrew Walker. *Java CAPS Basics: Implementing Common EAI Patterns,* 2008

[27] Xebia: *LIVRE BLANC: Comprendre et savoir utiliser un ESB dans une SOA*. Online en: http://www.xebia.fr

[28] Kim Haase: *Java Message Service API Tutorial*. Sun MicroSystems, 2002

[29] Gustavo Alonso, Fabio Casati, Harumi Kuno, Vijay Machiraju: *Web Services: Concepts, Architectures, Applications*. Springer, 2004

[30] Ganesh Prasad, Rajat Taneja, Vikrant Todankar: *Life about de service tier How to build applications front-ends in a service oriented world,* 2007

[31] Len Bass, Paul Clements, Rick Kazman: *Software Architecture in*

Practice, Second Edition. Addison Wesley, 2003

[32] http://www.itarchitect.org/articles/display.asp?id=137

[33] Martin Keen, Jonathan Bond, Jerry Denman, Stuart Foster, Stepan Husek, Ben Thompson, Helen Wylie: *Patterns: Integrating Enterprise Service Buses in a Service-oriented Architecture.* IBM RedBooks

[34] Heather Kreger: *Web Services Conceptual Architecture (WSCA 1.0).* IBM Software Group

[35] Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, Beth Stearns: *Designing Web Services with the J2EE 1.4 Platform JAX-RPC, SOAP, and XML Technologies.* Addison Wesley, 2004

[36] Gregor Hohpe: *Enterprise Integration Patterns - Asynchronous Messaging Architectures in Practice.* Conferencia JavaOne, 2004

[37] Duncan Scott, Michael Pecnik: *How to utilize enterprise information architecture to enable enterprise information integration.* Paper Factiva, 2003

[38] Douglas K. Barry: *Web services and service-oriented architecture: the sawy manager's guide.* Morgan Kaufmann, 2003

[39] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pal Krogdahl, Min Luo, Tony Newling: *Patterns: Service-oriented Architecture and Web Services.* IBM RedBooks, 2004

[40] Guido van Stormbroek: *Project Management In Open Source Communities,* 2007

[41] Karl Fogel: *Producing Open Source Software / How to Run a Successful Free Software Project,* 2005

[42] Varios Autores: *Copyleft Manual de uso.* Ed. Traficantes de sueños, 2006

[43] Ron Ten-Hove: *Using JBI for Service-Oriented Integration.* Sun Microsystems, 2006

[44] Dave Chappell: *Enterprise Service Bus.* O'Reilly, 2004. Online en: http://my.safaribooksonline.com/0596006756

[45] http://www.espaciosoa.net/2007/10/18/usos-practicos-de-un-esb

[46] http://www.infoq.com/presentations/Enterprise-Service-Bus

[47] http://www.theserverside.com/news/thread.tss?thread_id=35053

[48] http://www.misbytes.com/wp/2006/10/08/buses-de-servicios-empresariales-esb-soa-bpm-relacionando-todas-estas-siglas/

[49] David Pallmann: *Neuron ESB: An Enterprise Service Bus for the Microsoft Platform, parte 1.* Neudesic, 2006

[50] http://www.packtpub.com/article/aggregate-services-in-servicemix-jbi-esb

[51] http://www.thomas-bayer.com/soa-vs-eai-esb.htm

[52] Binildas C.A: *Service oriented Java Business Integration.* Pack Publishing, 2008

[53] Zhen Liu, Anand Ranganathan, Anton Riabov: *A planning-based approach for the automated configuration of the Enterprise Service Bus*. Service Oriented Computing ICSOC, 6th International Conference, Australia, 2008

[54] Especificación JBI: *Java Community Process.* http://jcp.org/en/jsr/detail?id=208

[55] Michael Wisler: *JBI based ESB as backbone for SOI applications*. 2007

[56] Karim R. Lakhani and Robert G Wolf: *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects*. MIT, 2005

[57] Foros de Discusión de Apache ServiceMix: http://servicemix.apache.org/discussion-forums.html

[58] Listas de correo de Apache ServiceMix: http://servicemix.apache.org/mailing-lists.html

[59] Bug Tracker en Apache ServiceMix:http://issues.apache.org/activemq/browse/SM

[60] Código Fuente de Apache ServiceMix: http://servicemix.apache.org/source.html

[61] FAQs de Apache ServiceMix: http://servicemix.apache.org/faq.html

[62] Documentación general de Apache ServiceMix: http://servicemix.apache.org/documentation.html

[63] Stoll, B. L.: *Fun and Software Development, Free / Open Source Research Community*. Http://opensource.mit.edu/online_papers.php. 2005

[64] Patrick E. Waterson: *The dynamics of work organization, knowledge and technology during software development*. International Journal of Human Computer Studies, Volume 46, 1997

[65] JDK (en su versión 1.6.0): http://java.sun.com

[66] Apache ServiceMix (en su versión 3.2.2): http://servicemix.org

[67] Apache Maven (en su versión 2.0.9): http://maven.apache.org

[68] Apache Lucene (en su versión 2.3.1): http://lucene.apache.org

[69] Apache XBean (en su versión 3.2): http://geronimo.apache.org/xbean

[70] Apache Log4J (en su versión 1.2.13): http://logging.apache.org/log4j

[71] Apache Camel (en su versión 1.4.0): http://activemq.apache.org/camel

[72] Spring Framework (en su versión 2.5.1): http://springframework.org

[73] Hibernate (en su versión 3.1.3): http://hibernate.org

[74] MySQL Java Conector (en su versión 5.0.5): http://mysql.com

[75] Compass (en su versión 2.0.0-RC2): http://compassframework.org

[76] Quartz (en su versión 1.5.2): http://www.opensymphony.com/quartz

[77] XStream (en su versión 1.1.2): http://xstream.codehaus.org

[78] Grails Framework (en su version 1.1): http://grails.org

[79] Groovy (en su versión 1.6): http://groovy.codehaus.org

[80] JUnit (en su versión 3.8.2): http://www.junit.org

[81] Emmanuel Bernard, John Griffin: *Hibernate Search In Action, Capítulo 1.* Manning, 2008

[82] Erik Hatcher and Otis Gospodnetić: *Lucene in Action, Capítulos 1, 2 y 3.* Manning, 2004

[83] Thomas Erl: *SOAWorld Magazine, Volume 8, Issue 6 – Introducing SOA Design Patterns,* 2008

[84] Arnon Rotem-Gal-Oz: *SOA Patterns*. Manning, 2007

[85] http://www.soapatterns.org

[86] Mike Gilpin: http://forrester.com/Research/Document/Excerpt/0,7211,35193,00.html, 2004

[87] http://eaipatterns.com